Alice and Bob: Reconciling Formal Models and Implementation

Almousa, Omar; Mödersheim, Sebastian Alexander; Viganò, Luca

Published in:
Programming Languages with Applications to Biology and Security

Link to article, DOI:
10.1007/978-3-319-25527-9_7

Publication date:
2015

Document Version
Peer reviewed version

Alice and Bob: Reconciling Formal Models and Implementation (Extended Version)*

Omar Almousa[1], Sebastian Mödersheim[1], and Luca Viganò[2]

[1] DTU Compute, Lyngby, Denmark
[2] Department of Informatics, King's College London, UK

Abstract. This paper defines the "ultimate" formal semantics for Alice and Bob notation, i.e., what actions the honest agents have to perform, in the presence of an arbitrary set of cryptographic operators and their algebraic theory. Despite its generality, this semantics is mathematically simpler than any previous attempt. For practical applicability, we introduce the language SPS and an automatic translation to robust real-world implementations and corresponding formal models, and we prove this translation correct with respect to the semantics.

1 Introduction

Alice-and-Bob notation is a simple and succinct way to specify security protocols: one only needs to describe what messages are exchanged between the protocol agents in an unattacked protocol run. However, it has turned out to be surprisingly subtle to define a formal semantics for such a notation, i.e., defining an inference system for how agents should compose, decompose and check the messages they send and receive. Such a semantics is necessary in order to automatically generate formal models and implementations from Alice-and-Bob specifications. However, even modeling messages in the free algebra, defining the semantics has proved far from trivial [11–13, 20, 22, 23]. To make matters worse, many modern protocols rely, for instance, on the Diffie-Hellman key agreement where the algebraic properties of modular exponentiation are necessarily part of the operational semantics, since the key exchange would be non-executable in the free algebra. For practical purposes, one can augment the semantics with support for just this special example like [27], but a general and mathematically succinct and rigorous theory is desirable.

We give in this work a semantics for an arbitrary set of operators and their algebraic properties. Despite this generality, the semantics is a much more succinct and mathematically simple definition than all the previous works (it fits on half a page) because it is based on a few general and uniform principles to define the behavior of the participants. This semantics was inspired by the similar

* This work was partially supported by the EU FP7 Projects no. 318424, "FutureID: Shaping the Future of Electronic Identity" (futureid.eu), and by the PRIN 2010-2011 Project "Security Horizons".

works of [24, 14], which we further simplify considerably. Our semantics is also subsuming the previous works in the free algebra and limited algebraic reasoning, as they are instances of our semantics for a particular choice of operators and algebraic properties (although this is not easy to show as explained below). We thus see our semantics as one of our main contributions since, from a mathematical point of view, a simple general principle that subsumes the complex definitions of many special cases is the most desirable property of a definition.[3]

This simple mathematical semantics, however, cannot be directly used as a translator from Alice-and-Bob notation to formal models or implementations since it entails an infinite representation and several of the underlying algebraic problems are in fact not recursive in general. We thus consider a particular set of operators and their algebraic properties that supports a large class of protocols, including modular exponentiation and multiplication. This theory not only subsumes the theories of previous papers, but also clarifies subtle details of the behavior of operators that were left implicit previously. For this theory, we define a *low-level* semantics that is much more complex than the mathematical *high-level* one but it is computable, and we formally prove that the low-level semantics is a correct implementation of the high-level one. The division into a simple mathematical high-level semantics as a "gold standard" and a low-level "implementable" semantics not only allows for a reasonable correctness criterion of the low-level semantics, but is in our opinion a major advantage over previous works that are a blending between mathematical and technical aspects.

To make our work applicable in practice, we have designed the *Security Protocol Specification language SPS* as a variant of existing Alice-and-Bob languages that contains many novel features valuable in practice. In particular, our notion of *formats* allows us to integrate the particular way of structuring messages of real-world protocols like TLS, rather than academic toy implementations; at the same time, we can use a sound abstraction of these formats in the formal verification. We have implemented the low-level semantics in a translator that can generate both formal models in the input languages of popular security protocol analysis tools (e.g., Applied π calculus in the syntax of ProVerif [10] or ASLAN for AVANTSSAR [5]) and implementations in JavaScript for the execution environment of the FutureID project (`www.futureid.eu`). We have demonstrated practical feasibility with a number of major and minor case studies, including TLS and the EAC/PACE protocols used in the German eID card.

We proceed as follows: we give the syntax of SPS in Section 2 and an extension of strands in Section 3. We define the semantics of SPS in Section 4 and discuss the connections from SPS to implementations and formal models in Section 5. In Section 6, we discuss related and future work, and conclude the paper.

[3] We have learned that from Pierpaolo Degano, who is renowned for his ability to explain complex things in a simple way.

2 SPS Syntax

In this section, we briefly introduce the syntax of SPS, which we will illustrate by referring to the example protocol specification in SPS given in Listing 1.1, in which two agents `A` and `B` use a symmetric key `shk(A,B)` to establish a fresh Diffie-Hellman key and securely exchange a `Payload` message.

```
Protocol: example
Types:
    Agent A,B;
    Number g, Payload, X, Y;
Mappings:
    shk: Agent,Agent-> SymmetricKey;
Knowledge:
    A: A, B, shk(A,B), g;
    B: A, B, shk(A,B), g;
Actions:
    A : Number X
    A -> B  : scrypt(shk(A,B), f1(A,B,exp(g,X)))
    B : Number Y
    B -> A  : scrypt(shk(A,B), f1(B,A,exp(g,Y)))
    A : Number Payload
    A -> B  : scrypt(exp(exp(g,Y),X), f2(Payload))
Goals:
    Payload secret of A,B
```

Listing 1.1. Example Protocol in APS

We give the syntax of SPS in EBNF, where we set all meta-symbols in blue and write Xs (for a non-terminal symbol X) to denote a comma-separated list X(, X)* of X elements; Const and Func are alphanumeric strings starting with a lower-case letter (e.g., `g` and `scrypt` in the example) and Var is an alphanumeric string starting with an upper-case letter (e.g., `X` in the example).

SPS ::= `Types` : (Type Idents;)*
`Mappings` : (Func : Types → Type;)*
`Formats` : (Func(Types);)*
`Knowledge` : (Role : Msgs;)* [`where` Role ≠ Role (& Role ≠ Role)*]
`Actions` : (Role Channel Role : Msg | Role : Type Var)*
`Goals` : (Role `authenticates` Role `on` Msg | Msg `secret of` Roles)*

Msg ::= Const | Var | Func(Msgs)
Ident ::= Const | Var | Func
Role ::= Const | Var
Type ::= `Agent` | `Number` | `PublicKey` | `PrivateKey` | `SymmetricKey` | `Bool` | `Msg`
Channel ::= [•] → [•]

We begin our explanation with the atomic elements: constants (Const) and variables (Var). One may think of the variables as parameters of a protocol description that must be instantiated for a concrete execution of the protocol; in our example, the variables `A` and `B` shall be instantiated with concrete agent names such as `a`, `b` or the intruder `p`[4], whereas `X` and `Y` should be instantiated with random numbers that are freshly chosen by `A` and `B`, respectively.

In the `Types` section, all constants and variables are declared with one of the pre-defined types, where the type `Msg` subsumes all types. By default, the

[4] We use `p` instead of `i` in honor of our "favorite intruder" Pierpaolo.

interpretation of SPS is *untyped*, i.e., types are used only by the SPS translator to check that the user did not specify any ill-typed terms. The types can however be used to generate a more restrictive typed model and under certain conditions this restriction is without loss of attacks [3]. The type `Agent` has a special relevance: constants and variables of this type we call *roles*, and the symbol ROLE in the above grammar must only be used for identifiers of type `Agent`. (This is an additional check we cannot directly express in a context-free grammar.)

While the semantics of Alice-and-Bob style languages that we give in the next section is generic for an arbitrary set of function symbols and their algebraic properties, the concrete implementation of SPS is for a set of fixed cryptographic function symbols. These are asymmetric and symmetric encryption (`crypt` and `scrypt`), digital signatures (`sign`), hash and keyed-hash functions (`hash` and `mac`), and modular exponentiation (`exp`) and multiplication (`mult`). There are of course corresponding operations for decryption and verification, but these are not part of an SPS specification; instead, their use is *derived* by the SPS translator according to the semantics in the next section.

In the `Mappings` section, one can specify a special kind of function symbols. These do not represent any actual operation that honest agents or the intruder can perform, but are used to describe the pre-existing setup of long-term keys. In our example, the mapping `shk` assigns to every pair of agents a unique value of type symmetric key; this is the easiest way to define shared keys for agents—including the intruder who will then share keys $\texttt{shk}(\texttt{p},\texttt{A})$ and $\texttt{shk}(\texttt{A},\texttt{p})$ with every other agent `A`. Public key infrastructures can be modeled in a similar way.

In the `Formats` section, one can specify a third kind of function symbols called *formats*. They abstractly represent how the concrete implementation structures the clear-text part of a message, such as XML-tags or explicit message-length fields. A format thus basically represents a concatenation of information, but in contrast to a plain concatenation operator as in other formal languages, the abstract format function symbols allow us to generate implementations with real-world formats such as TLS (see below). In the example, we have two formats: `f1` is used to exchange the Diffie-Hellman half-keys together with the agent names, and `f2` indicates the transmission of the `Payload` message. For simplicity, we model a payload message using a fresh random number `Payload`, representing a placeholder for an arbitrary message (depending on the concrete application); alternatively, this could be modeled using a mapping (e.g., $\texttt{payload}(\texttt{A},\texttt{B})$) that `A` knows initially and sends to `B` after the key establishment.

The three kinds of function symbols are thus: the cryptographic function symbols, the mappings and the formats. Except for the mappings, these are all *public*: all agents, including the intruder, can apply them to messages they know. Additionally, formats are *transparent*: every agent can extract the fields of a format. We can now build composed messages with these function symbols, where we assume the additional check that all SPS messages are well-typed (and are used with the proper arity). As typing is not essential for this paper, we do not discuss the details of the type expressions.

In the `Knowledge` section, we specify the *initial knowledge* of each of the protocol roles. This is essential as it determines how (and if) honest agents can execute the protocol. For instance, if in the example we were to omit the item $\texttt{shk}(\texttt{A},\texttt{B})$ in the knowledge of role `B`, then `B` could not decrypt the first message from `A` and thus not obtain `A`'s half key. Moreover, in the next step `B` would be unable to build the response message for `A`. Also, as we will define below, this specification indirectly determines the initial knowledge of the intruder: if a role is instantiated with `p`, then the intruder obtains the corresponding knowledge (in our case, all shared keys $\texttt{shk}(\texttt{A},\texttt{B})$ where $\texttt{A} = \texttt{p}$ or $\texttt{B} = \texttt{p}$). We require that all variables in the knowledge section be of type `Agent`. Finally, one can optionally forbid some instantiations of the roles, e.g., by the side condition $\texttt{A} \neq \texttt{p}$ or $\texttt{A} \neq \texttt{B}$.

The `Actions` section is the core of the specification: it specifies the messages that are exchanged between the roles. Additionally, we specify here explicitly when agents freshly create new values. In our example, `A` first creates the secret exponent `X` for the Diffie-Hellman exchange, computes the half-key $\texttt{exp}(\texttt{g},\texttt{X})$, inserts it into format `f1` and encrypts the message with the shared key $\texttt{shk}(\texttt{A},\texttt{B})$. To send this message, `A` uses the standard *insecure channel* (denoted with $\rightarrow$) on which the intruder can read, intercept, and insert messages arbitrarily. SPS also supports a notion of authentic, confidential, and secure channels as in [24], denoted with $\bullet\!\!\rightarrow$, $\rightarrow\!\!\bullet$ and $\bullet\!\!\rightarrow\!\!\bullet$, respectively. For instance, one may have specified the exchange of the half-keys without the encryption but using authentic channels where the intruder can see messages, but not insert messages except under his real name. This represents the *assumption* that the messages between `A` and `B` cannot be manipulated by an intruder, e.g., in device pairing of mobile devices, when `A` and `B` meet physically in a public place. The assumptions are reflected only in the formal model (by restricting the intruder behavior on such channels), while in the implementation it is the duty of the surrounding software module to connect a properly secured channel to the protocol module. One last point about the `Actions` section is that it shows the simplicity of an SPS specification, i.e., this section is very similar to the way one would informally describe a protocol in Alice and Bob notation.

In the final `Goals` section, we specify the *goals* the protocol aims to achieve. SPS provides built-in macros for the standard secrecy and authentication goals. In general, we instrument the description with events that reflect what is happening in the protocol execution, e.g., the event $\texttt{secret}(\texttt{A},\texttt{B},\texttt{Payload})$ reflects that `Payload` is supposed to be a secret between `A` and `B`. We then define attack states as predicates over these events. The events allow us to formulate security goals in a protocol-independent way rather than referring to the messages of the protocol.

3 Operational Strands

As a preparation for defining the SPS semantics, we first clarify the target language, i.e., we define an extension of the popular *strands* [28] that we call *operational strands*. Here we give in a glance the five extensions that we make. A

concrete example is shown in Fig. 1 and explained below and we give the formal details of operational strands in Appendix A.

First, send and receive steps can be annotated with a channel. Recall that SPS supports default insecure channels as well as authentic, confidential and secure ones. For the SPS semantics, this is only a label on the channels that is left unchanged in the translation; for the semantics of operational strands, the channels mean a restriction on the operations that the intruder can perform on the channel as explained in Appendix A. In textual representation, we write $\mathsf{send}(ch, t)$ and $\mathsf{receive}(ch, t)$ for sending and receiving message t over channel ch.

Second, we annotate each strand with the initial knowledge of the role it represents, denoted by a box above the strand (we define knowledge formally in Definition 2). The annotation has no meaning for the behavior of strands and is only needed during the translation process. In textual representation, we write the annotation with the knowledge M as M : *steps* at the beginning of the strand.

Third, recall that the original strand spaces are used to characterize sets of protocol executions and contain only ground terms. In contrast, we use them like a "light-weight" process calculus: terms may contain variables (representing values that are instantiated during the concrete execution). Also, we have the construct $\mathsf{fresh}\ X$ where the variable X will be bound to a fresh value. An important requirement is that operational strands are *closed* in the following sense: every variable must be *bound* by first occurring in the initial knowledge, in a fresh operation, in a macro (that we introduce shortly), or in a receive step. A bound variable must not occur subsequently in a fresh operation (i.e., it cannot be "re-bound"). In contrast, a bound variable may occur in a subsequent receive step, meaning simply that the agent expects the same value that the variable was bound to before.

Fourth, we extend strands with *events* (predicates over terms) to formulate security goals in a protocol-independent way. For instance, as we already remarked above, we may use the event $\mathsf{secret}(\mathtt{A}, \mathtt{B}, \mathtt{Payload})$ to express that message `Payload` is regarded as a secret between protocol roles `A` and `B`. Then we can define (independent of the concrete protocol) a violation of secrecy as a state where the intruder has learned `Payload` but is neither `A` nor `B`. We do not give here more details on goals, because from a semantical point of view we just treat the events as if they were messages on a special channel to a "referee" who decides if the present state is an attack; the handling of these events is uniform for a wide class of goals [3] and only limited by the abilities of current verification tools. In textual representation, we will simply write $\mathsf{event}(t)$ where t is a term characterizing the event.

Fifth, we add *checks* of the form $s \doteq t$. The meaning is that the agent can only continue if the terms s and t are equal and aborts otherwise. Also, we have *macros* of the form $\mathcal{X}_i := t$, which mean that we consider the same strand with all occurrences of $\mathcal{X}_i$ replaced by t. This is helpful for generating protocol

implementations, because the result of a computation t is stored in a variable $\mathcal{X}_i$ and does not need to be computed again later.

A formal definition of operational strands can be given as a process (interacting with a given environment). In Appendix A,we define a semantics as state-transition systems similar to [15], where a state $(S; K; E)$ consists of a set S of strands, a set K of messages that the intruder currently knows and a set E of events that have occurred. For instance, if S contains the strand $\mathsf{send}(\mathsf{insec}, t).rest$, where insec represents an insecure channel, then we can make the transition to a successor state where t is added to K and the send step is removed from the given strand.

4 SPS Semantics

Above we described the SPS syntax for a fixed set of cryptographic operators (for which we later give a fixed set of algebraic equations). In this section, we give a semantics that is parametrized over an *arbitrary* set of operators and algebraic properties, inspired by [24, 14]. One of the main contributions of our work is to give this general definition of a semantics for Alice-and-Bob style languages in a concise, mathematical way that is based on a few simple, general principles. The semantics is a function from SPS to (operational) strands; this function is in general not recursive because many of the underlying algebraic reasoning problems are not. The value of this general definition is its simplicity and uniformity: this is in fact the best mathematical argument why to define a concept in a particular way and not differently. In the next section, we then show that we can actually implement this semantics for the operators of SPS; in fact, we define a "low-level" semantics that is a computable function from SPS to strands (that is however so complicated that we give only an overview in this paper) and prove that it coincides with the general "high-level" semantics.

4.1 Message model

We define messages as algebraic terms and use the words *message* and *term* interchangeably. We distinguish two kinds of messages: (1) the *protocol messages* that appear in an SPS specification and (2) *labels* (or *recipes*) that are the messages in the strands the semantics translates to. It is necessary to make this distinction as the SPS specification reflects the ideal protocol run, while the semantics reflects the actual actions and checks that an honest agent performs in the run of the protocol. For the same reason, we will also distinguish between two kinds of variables: *protocol variables* and *label variables*.

Definition 1. *A* message model *is a four-tuple* $(\Sigma, V, \mathcal{L}, \approx)$. Σ *is a countable set of* function symbols*, all denoted by lower-case letters, where:* $\Sigma_0 \subseteq \Sigma$ *is a countable set of* constants*,* $\Sigma_p \subseteq \Sigma$ *is a finite set of* public operators *such as public-key encryption, and* $\Sigma_m \subseteq \Sigma$ *is a finite set of* mappings *(or* private operators*), disjoint from* Σ_p*. We assume a global public constant* $\top \in \Sigma_p \cap \Sigma_0$*.*

Table 1. Example of an equational theory $\approx$

(1) $\mathtt{dscrypt}(k,\ \mathtt{scrypt}(k,m)) \approx m$	(2) $\mathtt{vscrypt}(k,\ \mathtt{scrypt}(k,m)) \approx \top$
(3) $\mathtt{dcrypt}(\mathtt{inv}(k),\ \mathtt{crypt}(k,m)) \approx m$	(4) $\mathtt{vcrypt}(\mathtt{inv}(k),\ \mathtt{crypt}(k,m)) \approx \top$
(5) $\mathtt{open}(\mathtt{sign}(k,m)) \approx m$	(6) $\mathtt{vsign}(k, \mathtt{sign}(\mathtt{inv}(k),m)) \approx \top$
For every $\mathtt{f} \in \Sigma_f$ with arity n and for every $i \in \{1,\dots,n\}$	
(7) $\mathtt{get}_{i,\mathtt{f}}(\mathtt{f}(t_1,\dots,t_n)) \approx t_i$	(8) $\mathtt{verify}_{\mathtt{f}}(\mathtt{f}(t_1,\dots,t_n)) \approx \top$
(9) $\mathtt{exp}(\mathtt{exp}(t_1,t_2),t_3)) \approx \mathtt{exp}(t_1,\mathtt{mult}(t_2,t_3))$	(10) $\mathtt{mult}(t_1,t_2) \approx \mathtt{mult}(t_2,t_1)$
(11) $\mathtt{mult}(t_1,\mathtt{mult}(t_2,t_3)) \approx \mathtt{mult}(\mathtt{mult}(t_1,t_2),t_3)$	

V is a countable set of protocol variables. *$\mathcal{L} = \{\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \dots\}$ is a countable set of* label variables *disjoint from Σ and V. $\approx$ is a congruence relation over* ground terms *over Σ (i.e., terms without variables), which are denoted by $\mathcal{T}_\Sigma$. A* term *is thus a constant, a variable, or an application of a function (of Σ) on a term, and we write $\mathcal{T}_S(A)$ for the set of terms over signature S and variables from set A.*

As we define in a deduction relation below, the public operators in Σ_p are those functions that every agent and the intruder can apply to messages they know, i.e., the cryptographic operators (including operators for decryption that do not occur in the SPS specification) and the non-cryptographic formats. In contrast, the mappings in Σ_m are private, like `shk` in our example protocol that maps from two agents to their shared secret key, or `inv` that maps from public to private keys.

Example 1. As a concrete example of a message model that is representative for a large class of security protocols, let Σ_p contain all operators of the equations in Table 1, where $\approx$ is the least congruence relation satisfying the equations. For instance, `scrypt` represents symmetric encryption, `dscrypt` is the corresponding decryption operator and `vscrypt` is a *verifier*: given a term t and a key k, it tells us whether t is a valid symmetric encryption with key k. This models the fact that most symmetric ciphers include measures to detect when the decryption fails (e.g., when it is actually not an encrypted message or the given key is not correct) and in concrete implementations this verification will be part of the call to `dscrypt`. We emphasize that our message model explicitly describes such fine details that most security protocol analysis tools silently assume; we could similarly define a set of primitives that do not allow verification and the semantics will accordingly define which verifications honest agents can and cannot do.

Similarly, the operators `crypt`, `dcrypt` and `vcrypt` formalize asymmetric encryption, and `sign`, `open` and `vsign` formalize digital signatures.

Let $\Sigma_f \subseteq \Sigma_p$ be a set of formats declared in an SPS specification. Then, for each format $f \in \Sigma_f$ of arity n, $\mathtt{get}_{i,\mathtt{f}} \in \Sigma_p$ is an *extraction function* for the i-th field of the format (for all $1 \leq i \leq n$) and $\mathtt{verify}_{\mathtt{f}} \in \Sigma_p$ is a verifier to check that a given message has format `f`.

Moreover, we have `exp` and `mult` for modular exponentiation and multiplication as needed in many Diffie-Hellman-based protocols. As is often done, we

omit the modulus for ease of notation. Σ_p also contains `hash` and `mac` representing hash and keyed hash functions, respectively (`hash` and `mac` do not appear in Table 1 since they have no algebraic properties). Finally, a typical set of mappings could be: $\texttt{shk} : \texttt{Agent} \times \texttt{Agent} \to \texttt{SymmetricKey}$ to denote a shared key of two agents, $\texttt{pk} : \texttt{Agent} \to \texttt{PublicKey}$ for the public key of an agent, and $\texttt{inv} : \texttt{PublicKey} \to \texttt{PrivateKey}$ for the private key corresponding to a given public key. Although `pk` is typically publicly available, it should not be a public operator as it does not correspond to a computation that honest agents or the intruder can perform (rather the initial distribution of keys should be specified in the knowledge section of SPS). □

Definition 2. *A* labeled message t^l *consists of a* protocol message $t \in \mathcal{T}_\Sigma(V)$ *and a* label $l \in \mathcal{T}_{\Sigma_p}(\mathcal{L})$. *A* knowledge *is a substitution of the form* $M = [\mathcal{X}_1 \mapsto t_1, \ldots, \mathcal{X}_n \mapsto t_n]$, *where* $\mathcal{X}_i \in \mathcal{L}$ *and* $t_i \in \mathcal{T}_\Sigma(V)$. *We call the set* $\{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ *the* domain *of* M *and write* $|M| = n$ *for the* length *of* M. *We may also refer to* M *as a set of entries and write, e.g.,* $M \cup \{\mathcal{X}_j \mapsto t_j\}$ *to add a new entry (where* $\mathcal{X}_j$ *is not in the domain of* M*).*

Intuitively, the label variables represent *memory locations* of an honest agent. A label l is composed from label variables and public operators, and reflects what actions an honest agent has performed on elements of its knowledge. A labeled message t^l expresses that an honest agent performed the actions of l to obtain what the SPS specification represents by the term t. For instance, we represent the initial knowledge of `A` in Listing 1.1 by $[\mathcal{X}_1 \mapsto \texttt{A}, \mathcal{X}_2 \mapsto \texttt{B}, \mathcal{X}_3 \mapsto \texttt{shk}(\texttt{A},\texttt{B}), \mathcal{X}_4 \mapsto \texttt{g}]$ to express that `A` stores her name and `B`'s name in her memory locations $\mathcal{X}_1$ and $\mathcal{X}_2$, a key shared with `B` in $\mathcal{X}_3$, and the group `g` in $\mathcal{X}_4$.

4.2 Message Derivation and Checking

We now define how *honest* agents can derive terms from their knowledge. This is in the style of Dolev-Yao deduction relations, but extended to labeled messages to keep track of the operations that have been applied. The relation has the form $M \vdash t^l$ where M is a knowledge and t^l a labeled term.[5]

Definition 3. $\vdash$ *is the least relation that satisfies the following rules:*

$$\frac{}{M \vdash t^{\mathcal{X}_i}}\ Ax\,,\ [\mathcal{X}_i \mapsto t] \in M \qquad \frac{M \vdash t^l}{M \vdash s^m}\ Eq\,,\ s \approx t, l \approx m \qquad \frac{M \vdash t_1^{l_1}\ \ldots\ M \vdash t_n^{l_n}}{M \vdash f(t_1,\ldots,t_n)^{f(l_1,\ldots,l_n)}}\ Cmp\,,\ f \in \Sigma_p$$

The rule Ax expresses that an agent can deduce any message that it has in its knowledge, Eq expresses that deduction is closed under equivalence in $\approx$ (on terms and their labels), and Cmp allows agents to apply any public operator to deducible terms.

[5] One may employ an entirely different model for the intruder (e.g., a cryptographic one); using a Dolev-Yao style deduction for honest agents is simply the semantic decision that they perform only standard public operations (that would be part of a crypto API), but no operations that would amount to cryptographic attacks.

Example 2. As an example, consider again the algebraic theory of Table 1 and the knowledge $M = [\mathcal{X}_1 \mapsto \mathtt{k}, \mathcal{X}_2 \mapsto \mathtt{X}, \mathcal{X}_3 \mapsto \mathtt{scrypt}(\mathtt{k}, \mathtt{exp}(\mathtt{g}, \mathtt{Y}))]$. M contains three messages (or "memory locations") $\mathcal{X}_1, \ldots, \mathcal{X}_3$ that we associate with the corresponding messages of the SPS specification. We explain later how to reach a particular memory state, but for the intuition let us just consider an example scenario that would produce M for an agent A: the constant k could be part of the initial knowledge of A, $\mathtt{X}$ could be her secret Diffie-Hellman exponent, and the message stored in $\mathcal{X}_3$ could be what she received from another agent—*supposedly* the Diffie-Hellman half-key $\mathtt{exp}(\mathtt{g}, \mathtt{Y})$ encrypted with the key k. The tricky part here is that in general A will be unable to check that the received message has the correct form (i.e., that she did not receive just some garbage); it is part of the semantics to describe what A can check and what messages she will construct on the basis of the labels $\mathcal{X}_1, \ldots, \mathcal{X}_3$. Let us for instance consider the case that A should now—according to the SPS specification—generate the Diffie-Hellman full-key $t = \mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{X}), \mathtt{Y})$. That amounts to finding a label l such that $M \vdash t^l$, i.e., that *would* produce the Diffie-Hellman key, if the received message has the required form. Indeed, there is such a label as the following proof tree shows:

$$\dfrac{\dfrac{\dfrac{}{M \vdash \mathtt{X}^{\mathcal{X}_2}}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{M \vdash \mathtt{k}^{\mathcal{X}_1}}\,Ax \quad \dfrac{}{M \vdash \mathtt{scrypt}(\mathtt{k}, \mathtt{exp}(\mathtt{g}, \mathtt{Y}))^{\mathcal{X}_3}}\,Ax}{M \vdash \mathtt{dscrypt}(\mathtt{k}, \mathtt{scrypt}(\mathtt{k}, \mathtt{exp}(\mathtt{g}, \mathtt{Y})))^{\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)}}\,Cmp}{M \vdash \mathtt{exp}(\mathtt{g}, \mathtt{Y})^{\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)}}\,Eq}{M \vdash \mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{Y}), \mathtt{X})^{\mathtt{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)}}\,Cmp}{M \vdash \mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{X}), \mathtt{Y})^{\mathtt{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)}}\,Eq$$

In fact, we see the "recipe" to generate the term $\mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{X}), \mathtt{Y})$ in the label $\mathtt{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)$, i.e., A has to first apply decryption to term $\mathcal{X}_3$ using the term $\mathcal{X}_1$ as decryption key; if the received $\mathcal{X}_3$ message was indeed of the right form, this gives the other agent's half-key ($\mathtt{exp}(\mathtt{g}, \mathtt{Y})$ in SPS), and this is further exponentiated with $\mathcal{X}_2$ to supposedly yield the full key ($\mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{Y}), \mathtt{X})$ in SPS). Note that the semantics also tells us what happens if A in the actual execution receives some improper term for $\mathcal{X}_3$: she will simply apply the operations to it as prescribed and that may lead for instance to the protocol getting stuck (if nobody else can generate the key) or to an attack (if the intruder manages to find a term that breaks some security goals), or the garbage term may actually be detected by the checks on messages that we describe next, which in this example amounts to checking that the given term is indeed an encryption with the right key. □

The definition of the checks that honest agents can make on their knowledge is in fact based on the deduction relation $\vdash$. The checks will be written as equations between terms. To that end, we introduce the symbol $\doteq$ and define $\doteq$*-equations* as follows: an *interpretation* $\mathcal{I}$ is a total mapping from $\mathcal{L}$ to $\mathcal{T}_\Sigma(V)$ that we extend to a function from $\mathcal{T}_\Sigma(V \cup \mathcal{L})$ to $\mathcal{T}_\Sigma(V)$ as expected; then we define $\mathcal{I} \models s \doteq t$ iff $\mathcal{I}(s) \approx \mathcal{I}(t)$, and extend this to (finite or infinite) conjunctions of equations

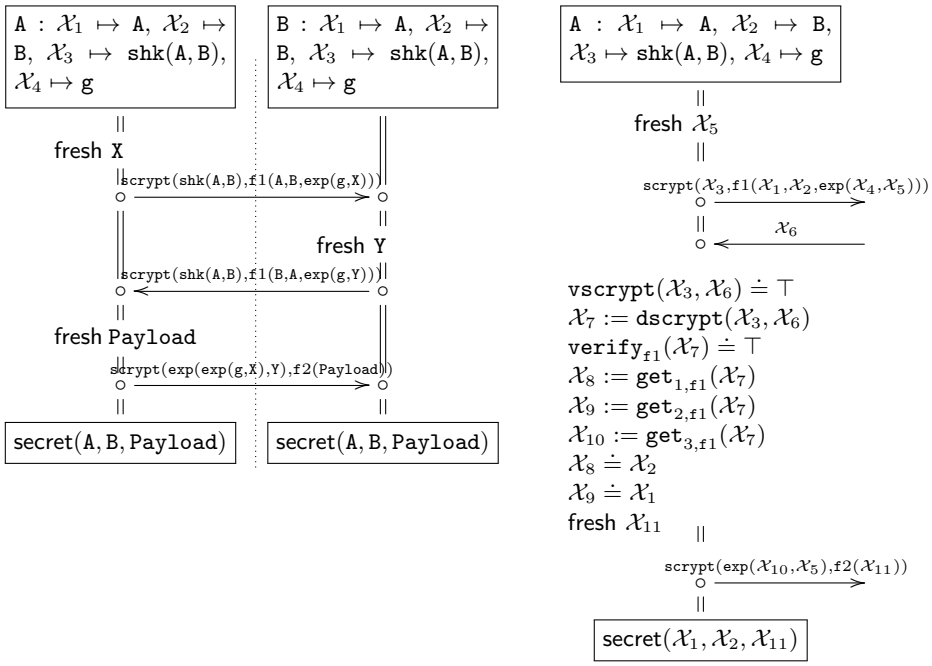

Fig. 1. (a) Example protocol (b) Operational strand of A

as expected. We define $\phi \models \psi$ iff $\mathcal{I} \models \phi$ implies $\mathcal{I} \models \psi$ for every interpretation $\mathcal{I}$; and $\phi \equiv \psi$ iff both $\phi \models \psi$ and $\psi \models \phi$.

Definition 4. *We define a* complete set of checks $ccs(M)$ *for a knowledge* M *as follows:* $ccs(M) = \bigwedge\{l_1 \doteq l_2 \mid \exists\, m \in \mathcal{T}_\Sigma(V).\, M \vdash m^{l_1} \wedge M \vdash m^{l_2}\}$.

$ccs(M)$ yields an infinite conjunction of checks that an agent can perform on his knowledge. Intuitively, $M \vdash m^{l_1}$ and $M \vdash m^{l_2}$ expresses that, according to the SPS specification, computing l_1 and l_2 *should* yield the same result m, and the agent can thus check that they actually do. For instance, consider $M = [\mathcal{X}_1 \mapsto k, \mathcal{X}_2 \mapsto \mathtt{hash}(m), \mathcal{X}_3 \mapsto \mathtt{scrypt}(k, m)]$. Amongst others, $ccs(M)$ then entails the checks $\phi = \mathtt{vscrypt}(\mathcal{X}_1, \mathcal{X}_3) \doteq \top \wedge \mathtt{hash}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)) \doteq \mathcal{X}_2$, i.e., the agent A can verify that $\mathcal{X}_3$ is an encryption and that $\mathcal{X}_2$ is the hash of the content of the encrypted message $\mathcal{X}_3$. Note that there are many more equations (e.g., $\mathcal{X}_1 \doteq \mathcal{X}_1$) and for every equation $s \doteq t$, we also have $h(s) \doteq h(t)$ for every unary public operator h. However, it holds that $ccs(M) \equiv \phi$, i.e., $ccs(M)$ is logically equivalent to ϕ and thus all other checks are redundant.

4.3 High-level Semantics

Now we can put everything together to define the semantics of SPS specifications by translation to operational strands. Fig. 1(a) shows our example protocol in the

style of message sequence charts. The first step towards an operational semantics is to split the protocol into different strands, one for each role, as indicated in Fig. 1(a) by the dotted line. We refer to the resulting strands as *plain strands*. Each plain strand shows how the protocol looks like from the point of view of that role in an ideal protocol run: what messages it is supposed to send and what messages it receives. The second step towards the operational semantics is to identify the precise set of actions, i.e., how messages are composed or decomposed, and what checks need to be performed on received messages. Fig. 1(b) shows how this operational description looks like for role `A` of the example (role `B` is very similar). Now we define the *high-level semantics* as a function $[\![\cdot]\!]_H$ (with initial case $[\![\cdot]\!]_{H_0}$) that maps from plain strands like (a) to the operational strands like (b).

In a nutshell, we use the labeled deduction $M \vdash t^l$ to define how an agent composes an outgoing message (or event), and we use the *ccs* function whenever an agent receives a new message, formalizing the set of checks that the agent can perform at this point. Note that this is an infinite conjunction and we later show how to obtain an equivalent finite conjunction for the example theory.

Definition 5. $[\![\cdot]\!]_H$ *translates from plain to operational strands as follows:*

$$
\begin{array}{ll}
[\![M : \mathit{strand}]\!]_{H_0} & = M : \mathit{ccs}(M).[\![\mathit{strand}]\!]_H(M) \\
[\![\mathsf{receive}(\mathit{ch}, t).\mathit{rest}]\!]_H(M) & = \mathsf{receive}(\mathit{ch}, \mathcal{X}_{|M|+1}).\mathit{ccs}(M \cup [\mathcal{X}_{|M|+1} \mapsto t]). \\
 & \quad\ [\![\mathit{rest}]\!]_H(M \cup [\mathcal{X}_{|M|+1} \mapsto t]) \\
[\![\mathsf{send}(\mathit{ch}, t).\mathit{rest}]\!]_H(M) & = \mathsf{send}(\mathit{ch}, l).[\![\mathit{rest}]\!]_H(M) \text{ where } M \vdash t^l \text{ for some label } l \\
[\![\mathsf{event}(t).\mathit{rest}]\!]_H(M) & = \mathsf{event}(l).[\![\mathit{rest}]\!]_H(M) \text{ where } M \vdash t^l \text{ for some label } l \\
[\![\mathsf{fresh}\ X.\mathit{rest}]\!]_H(M) & = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![\mathit{rest}]\!]_H(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\}) \\
[\![0]\!]_H(M) & = 0
\end{array}
$$

The first rule initializes the translation, by computing the checks that can be made on the initial knowledge of the strands. The second rule says that each received message is associated with a new label variable $\mathcal{X}_{|M|+1}$ in the agent's knowledge and afterwards we use *ccs* to generate all checks that the agent can perform on the augmented knowledge. The third rule is for sending the SPS protocol message t. Here we use the relation $M \vdash t^l$ to require that the agent can generate the required term t from the current knowledge M using the concrete sequence of actions l; this is explained in more detail below. The event rule is very similar to sending. The fifth rule translates the construct $\mathsf{fresh}\ X$: we simply pick a new label variable $\mathcal{X}_{|M|+1}$ that will store the fresh value in the translated strand, and bind it in the knowledge to the protocol variable X. The final rule is straightforward.

Let us continue Example 2, where we considered an agent with knowledge $M = [\mathcal{X}_1 \mapsto \mathtt{k}, \mathcal{X}_2 \mapsto \mathtt{X}, \mathcal{X}_3 \mapsto \mathtt{scrypt}(\mathtt{k}, \mathtt{exp}(\mathtt{g}, \mathtt{Y}))]$. (As explained above, this may result from a strand that initially knows a key in $\mathcal{X}_1$, has freshly generated an exponent $\mathcal{X}_2$, and has received the message $\mathcal{X}_3$.) Suppose that the next step is $\mathsf{send}(\mathsf{insec}, \mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{X}), \mathtt{Y}))$ (in fact, in a more realistic example, it would be a message encrypted with this term as a key). The semantics tells us to determine any label l such that $M \vdash \mathtt{exp}(\mathtt{exp}(\mathtt{g}, \mathtt{X}), \mathtt{Y})^l$, which is possible for the label $l = \mathtt{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)$ as shown in the example previously. Thus, the

translation can be in this case $\mathsf{send}(\mathsf{insec}, \mathtt{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2))$. Note that we said "*can*" here, because there are other labels, e.g., any label l' such that $l \approx l'$.

More generally, given M and t, there is in general not a unique l such that $M \vdash t^l$. First, consider the case that there is no such l. In this case, the agent has no means (within the deduction relation) to obtain the term t from its current knowledge. We thus say the protocol is *non-executable* and its semantics is undefined. This executability check is an important sanity check on SPS specifications, ensuring that all steps of the protocol can actually be performed at least when no intruder is interfering and the network does not loose messages. Other formal specification language like Applied π that specify the different roles separately as processes cannot have such an executability check, because unlike SPS, there is no formal relationship between the messages that one role is sending and another is receiving. Thus, if a modeler accidentally specifies messages slightly differently in two processes, they may be unable to communicate and get stuck in their execution; then a flawed protocol may be trivially verified as secure because of the specification mistake. The executability check in SPS drastically reduces the chance of such mistakes.

Second, if there is a label l, then there will typically be infinitely many of them (trivially by performing redundant encryptions and decryptions). Our semantics does not prescribe which of the labels has to be taken (and the implementation below will take in some sense the simplest one). A key insight is that this does not make the semantics ambiguous: if $M \vdash t^{l_1}$ and $M \vdash t^{l_2}$ then $ccs(M) \models l_1 \doteq l_2$. Thus, since we always perform the checks on the knowledge after each received message, we know that the choice of labels does not make a difference.

As an example, observe that the operational strand we have given in Fig. 1(b) for our example protocol is correct according to this semantics (when resolving the $\mathcal{X} := t$ macros): all outgoing messages have an appropriate label (for which $M \vdash t^l$ holds), and all checks $s \doteq t$ do indeed logically follow from $ccs(M)$ for the respective M. In fact, we claim that the checks are logically equivalent to $ccs(M)$, i.e., all other checks are redundant; it is part of the results of the next section to prove that and derive the given checks automatically.

We emphasize the succinctness of the definitions: Definitions 2–5 together fit on half a page and yet we define the semantics for an arbitrary set of cryptographic operators and algebraic properties. We believe that this is the best argument that the semantics of Alice-and-Bob notation should be defined this way—deriving from simple, general, uniform principles. However, this simple semantics cannot be directly used as a translator from Alice-and-Bob notation to formal models or implementations as it entails an infinite representation and several of the underlying algebraic problems are in fact not recursive in general.

Theorem 1 *For a given strand S, the problem to compute a finite representation of $[\![S]\!]_H$, if it exists, is not recursive.*

Proof Sketch. It is immediate that $\vdash$ is in general an undecidable relation (take an undecidable $\approx$). Similarly, the relation $\{(M, s, t) \mid ccs(M) \models s \doteq t\}$ is un-

decidable. It follows that for given a knowledge M, the problem to compute a finite conjunction ϕ, such that $\phi \equiv ccs(M)$, if one exists, is not recursive. □

4.4 Implementing the Semantics

Despite this general undecidability result, for a special theory we can give a more low-level, procedural semantics that is actually computable and prove that it correctly implements the high-level semantics. More specifically, we now sketch how to actually compute the semantics for the example theory in Table 1 and where we additionally require that the SPS specification (and thus the plain strands) does not contain any destructors or verifiers.

Theorem 2 *For our example theory in Table 1, for every strand S in which no destructors or verifiers occur, $[\![S]\!]_H$ can be finitely represented and it is recursive.*

Implementation/Constructive Proof. First we split the problem into a *constructor* and a *destructor/verifier* part (note they are not independent, e.g., in order to decrypt a message one may need to first compose a key). We also split the example theory into (i) equations C that describe destructors and verifiers (the first 8 equations in the Table 1) and (ii) equations F that just "rearrange" terms (the remaining equations). We then use equations C as rewrite rules and apply them modulo F (working on F-equivalence classes); the resulting rewrite relation $\rightarrow_{C/F}$ is convergent and we consider only normalized terms.

For our example theory (Table 1), we define two functions, $compose_M(t)$ and $analyze(M, \varphi)$. First, $compose_M(t)$ implements the "constructor" part of the $\vdash$ relation: find all labels l such that $M \vdash t^l$ when using only constructors of Σ_p (no destructors and verifiers) and using only equations from F. Note that the set of such labels l is always finite. Second, $analyze(M, \varphi)$ starts with a knowledge M and a set of checks φ that have already been performed (so they do not need to be checked again). It computes a pair (M', φ'). Here, M' is an *analyzed* extension by all subterms that can be obtained by applying destructors and normalizing the result; for this purpose, *analyze* calls the *compose* function to compose decryption keys when necessary. Also, for each decryption, the analysis will produce as part of φ' a new macro $\mathcal{X}_i := l$, where $\mathcal{X}_i$ is the label variable in the augmented knowledge that holds the result of the decryption and l is the recipe for obtaining it. Similarly, for each such decomposition step, we have a check from the respective verifier that is also added to φ'. Further, *analyze* will check for every term whether there is a different way to compose it (using again the *compose* function) and, if so, generate the according checks. Finally, for all pairs of terms where the root operator is `mult` (and analogously for `exp`), we must check if the least common multiple can be generated from each of them. For instance, knowing $[\mathcal{X}_1 \mapsto ab, \mathcal{X}_2 \mapsto ac, \mathcal{X}_3 \mapsto b, \mathcal{X}_4 \mapsto c]$, we can derive the check $\mathcal{X}_1\mathcal{X}_4 \doteq \mathcal{X}_2\mathcal{X}_3$.

We then show that for an analyzed knowledge, every derivable term can be derived using only *compose* and the checks resulting from *analyze* are equivalent to those of *ccs* modulo resolving the macros that *analyze* generates. Based on

this, we obtain the following computable low-level semantics that translates from plain strands to operational strands and mirrors the structure of the high-level semantics:

$$
\begin{array}{ll}
[\![M : \mathit{strand}]\!]_{L_0}(\emptyset, \top) & = M : \varphi.[\![\mathit{strand}]\!]_L(M', \varphi) \text{ where } (M', \varphi) = \mathit{analyze}(M, \top) \\
[\![\mathsf{receive}(ch, t).\mathit{rest}]\!]_L(M, \varphi) & = \mathsf{receive}(ch, \mathcal{X}_{|M|+1}).\varphi'.[\![\mathit{rest}]\!]_L(M', (\varphi \wedge \varphi')) \\
 & \qquad \text{where } (M', \varphi \wedge \varphi') = \mathit{analyze}(M \cup [\mathcal{X}_{|M|+1} \mapsto t], \varphi) \\
[\![\mathsf{send}(ch, t).\mathit{rest}]\!]_L(M, \varphi) & = \mathsf{send}(ch, l).[\![\mathit{rest}]\!]_L(M, \varphi) \text{ where } l \in \mathit{compose}_M(t) \\
[\![\mathsf{event}(t).\mathit{rest}]\!]_L(M, \varphi) & = \mathsf{event}(l).[\![\mathit{rest}]\!]_L(M, \varphi) \text{ where } l \in \mathit{compose}_M(t) \\
[\![\mathsf{fresh}\ X.\mathit{rest}]\!]_L(M, \varphi) & = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![\mathit{rest}]\!]_L(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\}, \varphi) \\
[\![0]\!]_L(M, \varphi) & = 0
\end{array}
$$

The full details of *compose* and *analyze* and the proofs of their correctness can be found in Appendix B. Based on this, we also prove that the two levels of our semantics ($[\![\cdot]\!]_H$ and $[\![\cdot]\!]_L$) coincide, i.e., given the same plain strand as input, they produce equivalent operational strands. □

5 Translations from Operational Strands

We now come to the "last mile" of the translation: to translate operational strands into an actual implementation and into a formal model for automated verification. Fig. 2 shows this translation for the role `A` of our example in Fig. 1(b); as target languages we have here JavaScript for protocol implementations and Applied π for the formal model.[6]

One can easily see a very close correspondence between the two translations: roughly, they both use the same operators in the same way, only in the formal model they are function symbols in an "abstract" term algebra, whereas in the implementation they are *corresponding* API calls. It is one of the contributions of this work to achieve such a close correspondence. While the use of crypto-APIs is of course standard, our notion of formats extends this API idea also to the non-cryptographic operations: all the technical details of parsing and pretty-printing are hidden in the classes for the given formats. Of course, just like the crypto-API, also the "non-crypto-APIs" require a robust implementation (that does not suffer from buffer overflows, for instance), but we want to argue that our setup with APIs is a suitable way to "cut the cake".

The close correspondence allow us to argue that there is no systematic discrepancy between formal model and implementation, if the function symbols have the corresponding meaning—but that is indeed subtle. Comparing the translation with the input strand of Fig. 1(b), there are only two significant differences: all the explicit verifiers of the strands are removed and the implementation does not contain events; besides that, the translation is mainly adapting to the syntax of the target language. For this reason, we do not give here a

[6] One may argue that JavaScript is not suitable for implementing security protocols, but in fact, using systematic mechanisms such as our formats, we can produce robust implementations that do not suffer from type flaw attacks, for instance. It is relatively easy to adapt to other languages like Java or the AVANTSSAR Platform [5], e.g., for using the tool OFMC, for which we have implemented a connector.

formal definition of the translation functions to JavaScript and Applied π (that can be found in Appendix C), but only discuss a few interesting aspects.

5.1 Experimental Results

The translator has been implemented as part of the FutureID project and is available at [**?**]. In the project, we have considered several real-world case studies such as the TLS handshake [16] as one of the most widely used protocols, the protocols EAC and PACE [19] that are used by the German eID card, and 30 smaller protocols. In particular, for our main case studies TLS, EAC and PACE, we did implement the precise message formats of the standards [18]. As part of FutureID, an execution environment has been defined that invokes the JavaScript code with suitable values for the parameters [17]. For the formal verification, we have used our case studies to check that ProVerif finds the known attacks in the small examples and verifies all other protocols. The entire test suite runs in less that 11 seconds on a 2.67 GHz machine.

5.2 JavaScript Translation

Crypto API. We of course rely on the execution environment to have suitable implementations of the cryptographic primitives, e.g., the `exp` operator will in fact be mapped to elliptic curve cryptography. We assume that the call `dscrypt`(k, m) will fail (aborting execution) if m is not a message encrypted with key k. This is why we do not include verifier checks in this translation. For simplicity, we omitted the optional annotation of primitives with the precise algorithm and key length (that is only necessary when using different ones in the same protocol).

Formats. The notion of formats allows us to integrate the actual message formats of real-world protocols like TLS. Similar to the cryptographic operators, we also rely on an API and implementation of non-cryptographic operators: for each format declared in the specification, we require a Java class that basically contains a parser and a pretty printer for that format (a.k.a. serialization/deserialization). For the example format `f1` the class `f1` must have three member variables of type byte string to represent the three fields of the form (as raw data). It must have two constructors: the first takes three strings as input and just stores them in the member variables (cf. the first `new f1` in the example), the second takes a single string and tries to parse it as format `f1`, and this may fail (cf. the second `new f1` in the example). Further, we have the `get`i() functions to obtain the i-th field and `encode`() to output a string. For a more detailed discussion of formats and TLS see [26].

5.3 Applied π Translation

Algebraic Properties. Let us start with the most subtle problem: the algebraic properties of the cryptographic and non-cryptographic operators. We can express

```
function proc_A(X1,X2,X3,X4,ch){
 Number  X5 = genNumber();
 ch.send(scrypt(X3,new f1(X1,X2,
   exp(X4,X5)).encode()));
 var X6 = ch.receive();
 var X7 = dscrypt(X3, X6);
 f1 X7a = new f1(X7);
 var X8 = X7a.get1();
 var X9 = X7a.get2();
 var X10 = X7a.get3();
 if(X8 != X2) error();
 if(X9 != X1) error();
 Number  X11 = genNumber();
 ch.send(scrypt(exp(X10,X5),
    new f2(X11).encode()));

}
```

```
let proc_A(x1,x2,x3,x4:bitstring,ch:Chann)=
  new x5:bitstring;
  out(ch,scrypt(x3,f1(x1,x2,
    exp(x4,x5))));
  in(ch,x6:bitstring);
  let x7:bitstring = dscrypt(x3,x6) in

  let x8:bitstring = f1get1(x7) in
  let x9:bitstring = f1get2(x7) in
  let x10:bitstring = f1get3(x7) in
  if(x8 = x2) then
  if(x9 = x1) then
  new x11:bitstring;
  out(ch,scrypt(exp(x10,x5),
    f2(x11)));
  event secret(x1,x2,x11);
  0.
```

Fig. 2. Translation to JavaScript and Applied π Calculus of role `A` of the example

cancelation, e.g., `reduc forall m,k : bitstring; dscrypt(k, scrypt(k,m)) = m.` (and the translator will automatically generate corresponding rules for the get-functions of the declared formats). However, during the verification process of ProVerif, where processes get translated into Horn clauses, these destructors get encoded into pattern matching—in the Horn clauses occur no destructors or verifiers. This transformation corresponds to an implicit verifier: in our example, the `let x7` clause will fail if the message `x6` is not of the form `scrypt(x3,·)`. Thus, also the ProVerif translation does not have verifiers. While this is expressing the algebraic theory we want at this point, directly formulating the equations for `exp` and `mult`, ProVerif will not terminate. For standard Diffie-Hellman, it is sound to restrict ourselves to the following equation that works with ProVerif [21, 25]:

$$\texttt{equation forall x,y : bitstring; exp(exp(g,x),y) = exp(exp(g,y),x)}\,.$$

The translator can only give a warning when the SPS specification is outside the fragment for which the soundness result holds.

Process Instantiation. We formulate all possible instantiations of the protocol: every role can be played by any agent, including the intruder, and we want to allow for any number of sessions of the protocol in parallel. It is not trivial to specify this manually, but the SPS compiler offers a systematic way to generate the instantiation. Recall that the initial knowledge of each role in the SPS specification can only contain variables of type `Agent` and long-term keys have to be specified using functions like `shk`. This allows us to instantiate the knowledge for any value of the role variables. For our example, we have the following specification (where the free name `pub` represents an insecure channel):

```
process
!new x:bitstring;out(pub,x)|
  !in(pub,(b:bitstring));proc_A(x,b,shk(x,b),g,pub)|
```

```
  out(pub,(p,b,shk(p,b),g))|
!in(ch,(a:bitstring));proc_B(a,x,shk(a,x),g,pub)|
  out(pub,(a,p,shk(a,p),g))
```

The first replication operator generates an unbounded number of honest agent names (in variable `x`) that are broadcast on `pub`. Then we generate an unbounded number of instances of `proc_A` for each `x` and each name `b` that we receive from the public channel (thus, the intruder can choose who will play role `B`). We also output on `pub` the initial knowledge that the intruder needs for playing role `A` under his real name `p`. The last two lines are similar for role `B`.

6 Conclusions and Related Work

The formal definition of languages based on the Alice-and-Bob notation requires one to identify the concrete set of actions that honest agents have to perform, which is relevant both for a formal model for verification and for generating implementations. Previous works have proposed fairly involved deduction systems for this purpose and there is no (even informal) justification why these systems would be suitable definitions. Our high-level semantics $[\![\cdot]\!]_H$, inspired by [24, 14], gives a mathematically succinct and uniform definition of Alice-and-Bob notation following a few general principles, and at the same time it supports an arbitrary set of operators and algebraic properties. The succinctness and generality is, in our opinion, a strong argument for this semantics as a standard. As $[\![\cdot]\!]_H$ entails problems that are not recursively computable in general, we defined the low-level semantics $[\![\cdot]\!]_L$ for a particular theory and proved its correctness with respect to $[\![\cdot]\!]_H$. While $[\![\cdot]\!]_L$ is similar (and similarly involved) as previous definitions of semantics for the Alice-and-Bob notation [22, 23, 13, 12, 20, 7], we are the first to give a complete formal treatment of the key algebraic properties for destructors, verifies, exponentiation and multiplication.[7]

With respect to other implementation generators like [29, 27], our key improvements are as follows. First, we give a uniform way to generate both formal models and implementation from the operational strands, ensuring a one-to-one correspondence between them. Second, replacing the abstract concatenation operator from formal models with formats allows us to generate code for any real-world structuring mechanism like XML formats or TLS-style messages. The only work that provides similar features is [6], which however starts at the π calculus level, comparable to the output of our low-level semantics. In reference to works that consider the verification of the actual implementation source code like [8], we agree with [9] that the converse problem, i.e., turning formal models into code like in this paper, is harder. However, in the case of SPS this extra effort takes a large part of the burden off the user, i.e., SPS carries the task

[7] Interestingly also the Festschrift for José Meseguer this year received a treatment of Alice and Bob notation [7] that is very similar to our low-level semantics $[\![\cdot]\!]_L$, however cannot handle exponentiation and multiplication. Thus, we can conclude that Pierpaolo received a strictly stronger Festschrift.

of formally verifiable implementations to a higher level of abstraction without suffering from flaws that are abstracted away in the formal model.

Finally, we point out a strong similarity between our notion of knowledge and the notion of *frames* in Applied π calculus [2]. We allow ourselves minor deviations from the frame concept, in particular not using *name restrictions*; instead, constants are by default not public in our setting. This makes the treatment in this paper easier but does not fundamentally change the concept (or its expressive power). For what concerns existing decision results for frames, the deduction relation $\vdash$ has been studied, e.g., in [1]. It is known that deduction is decidable for convergent subterm theories (like our equations (1)–(8)) and that disjoint associate-commutative operators as in (9)-(11) can easily be combined with it. Many results consider the static equivalence of frames which is interesting for privacy properties, namely whether the intruder is able to distinguish two frames ("knowledges"). In the SPS semantics, we have a substantially different problem to solve: we have only one knowledge M (and it is the knowledge of an honest agent) and we need to finitely characterize $ccs(M)$, i.e., what checks the agent can make on M to ensure that all received messages have the required shape. This indeed has some similar traits to static equivalence: also here one has to check pairs of recipes (albeit with respect to two frames). Despite this similarity, the problems are so different that it seems not directly possible to reuse decision procedures for static equivalence for computing $ccs(M)$. Moreover, our `exp`/`mult` theory is not yet supported in static equivalence results. A further investigation and generalization, namely with inverses for `mult`, is part of our ongoing research.

References

1. M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci.*, 367(1-2):2–32, 2006.
2. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In C. Hankin and D. Schmidt, editors, *POPL*, pages 104–115. ACM, 2001.
3. O. Almousa, S. Mödersheim, P. Modesti, and L. Viganò. Typing and Compositionality for Security Protocols: A Generalization to the Geometric Fragment. In *ESORICS 2015*, 2015. To appear, Available at `http://compute.dtu.dk/~samo`.
4. O. Almousa, S. Mödersheim, and L. Viganò. Alice and Bob: Reconciling Formal Models and Implementation (Extended Version). Technical report, DTU Compute, 2015. Available at `http://www.imm.dtu.dk/~samo/`.
5. A. Armando et al. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS*, 2012.
6. M. Backes, A. Busenius, and C. Hritcu. On the development and formalization of an extensible code generator for real life security protocols. In *NFM*, 2012.
7. D. Basin, M. Keller, S. Radomirović, and R. Sasse. Alice and Bob Meet Equational Theories. In *Festschrift for Jos´e Meseguer on his 65th Birthday*, 2015.
8. K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Verified Cryptographic Implementations for TLS. *ACM Trans. Inf. Syst. Secur.*, 15, 2012.
9. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSF 19*, pages 139–152. IEEE, 2006.

10. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSF*, pages 82–96. IEEE, 2001.
11. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.
12. S. Briais and U. Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.
13. C. Caleiro, L. Viganò, and D. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theoretical Computer Science*, 367(1):88–122, 2006.
14. Y. Chevalier and M. Rusinowitch. Compiling and securing cryptographic protocols. *Information Processing Letters*, 110(3):116–122, 2010.
15. C. Cremers and S. Mauw. Operational semantics of security protocols. In *Scenarios: Models, Transformations and Tools*, 2005.
16. T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol, Version 1.2, 2008.
17. FutureID Project. Deliverable D42.6: Specification of execution environment, 2014. `www.futureid.eu`.
18. FutureID Project. Deliverable D42.8: APS Files for Selected Authentication Protocols, 2015. `www.futureid.eu`.
19. German Federal Office for Information Security (BSI). Advanced Security Mechanism for Machine Readable Travel Documents, 2008. Available at `www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110`.
20. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *LPAR 2000*, pages 535–554. Springer, 2000.
21. R. Küsters and T. Truderung. Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation. In *CSF*, pages 157–171. IEEE, 2009.
22. G. Lowe. Casper: A compiler for the analysis of security protocols. In *CSFW*, pages 18–30. IEEE, 1997.
23. J. Millen. CAPSL: Common authentication protocol specification language. Technical report, Technical Report MP 97B48, The MITRE Corporation, 1997.
24. S. Mödersheim. Algebraic Properties in Alice and Bob Notation. In *ARES'09*, pages 433–440. IEEE, 2009.
25. S. Mödersheim. Diffie-Hellman without Difficulty. In *FAST*, pages 214–229, 2011.
26. S. Mödersheim and G. Katsoris. A sound abstraction of the parsing problem. In *CSF*, pages 259–273. IEEE, 2014.
27. P. Modesti. Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx. In *STM*, 2014.
28. F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999.
29. B. Tobler and A. C. Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Service*. Springer, 2005.

A Operational Strands

A.1 The Syntax of Operational Strands

The syntax of operational strands is a slight extension of the well-known strand spaces:

$$\begin{aligned}\textsc{Strand} ::= \textsc{Knowledge}: (\ &\mathsf{send}(\textsc{Channel}, \textsc{Msg}).\ |\ \mathsf{receive}(\textsc{Channel}, \textsc{Msg}).\\ |\ &\mathsf{event}(\textsc{Msg}).\ |\ \textsc{Msg} \doteq \textsc{Msg}.\ |\ \textsc{Var}:=\textsc{Msg}.\ |\ \mathsf{fresh}\ \textsc{Var}.\ )^*\ 0\end{aligned}$$

Note that in plain strands, no equalities occur. The non-terminals Channel, Msg, and Var are as in the SPS syntax. Knowledge, typically denoted by M in concrete strands, stands for a knowledge as defined in Definition 2, i.e., a substitution from label variables to protocol terms. We may omit this knowledge prefix of an operational strand when not relevant, as it is mainly used as an annotation in the semantics of SPS.

We define the free variables of an operational strand as follows:

$$\begin{aligned} fv(M : rest) &= fv(rest) \setminus dom(M)\\ fv(\mathsf{send}(ch, t).rest) &= fv(ch) \cup fv(t) \cup fv(rest)\\ fv(\mathsf{receive}(ch, t).rest) &= (fv(rest) \setminus fv(t)) \cup fv(ch)\\ fv(\mathsf{event}(t).rest) &= fv(t) \cup fv(rest)\\ fv(s \doteq t.rest) &= fv(s) \cup fv(t) \cup fv(rest)\\ fv(x := t.rest) &= (fv(rest) \setminus \{x\}) \cup fv(t)\\ fv(\mathsf{fresh}\ x.rest) &= fv(rest) \setminus \{x\}\\ fv(0) &= \emptyset\\ fv(x) &= \{x\}\\ fv(f(t_1, \ldots, t_n)) &= fv(t_1) \cup \ldots \cup fv(t_n)\end{aligned}$$

We require that all operational strands are *closed*, i.e., all variables, before being "used", are bound by occurring in the knowledge, in a received message, or in a fresh step. Further, a bound variable cannot occur in a fresh step (e.g., $\mathsf{fresh}\ x.\mathsf{fresh}\ x.0$ is not allowed) or a macro (e.g., $x := x$ cannot occur in a strand, since then x is bound earlier, violating that it cannot be re-bound, or x is a free variable of the strand). When a bound variable occurs in a receive step, it is not "re-bound", i.e., $\mathsf{receive}(ch, x).\mathsf{receive}(ch, x).rest$ by the following semantics will be equivalent to $\mathsf{receive}(ch, x).\mathsf{receive}(ch, y).x \doteq y.rest$.

A.2 The Semantics of Operational Strands

Similar to [15], we define the semantics of operational strands as an infinite-state transition system, where a state $(S; K; E)$ consists of (1) a set S of closed strands, (i.e., every variable occurs first in a receive message, in a macro, or in a creation of a fresh value), (2) a set K of messages (the intruder knowledge), and (3) a set E of events that have occurred. This transition system is defined by an initial state and a transition relation.

The initial state Recall that in an SPS specification, only variables of type agent may be used in a knowledge declaration; therefore the co-domain of the knowledge M of each operational strand of the protocol will only contain such agent-typed variables. The first step in defining the semantics is to consider all possible instantiations of these agent variables with concrete agent names; and create infinitely many copies of these operational strands to model an unbounded number of sessions between any agents.

Let therefore $\mathfrak{S} = \{s_1, \cdots, s_k\}$ be the set of operational strands of a protocol, one for each role of the protocol. Let us further denote by R_i the name of the role (i.e., a constant or variable of type agent) that is described by the operational strand s_i, M_i be the knowledge of s_i and $steps_i$ be the steps of s_i, i.e., $s_i = M_i : steps_i$. Let Ag be a countably infinite set of constants of type `Agent`, including `p` denoting the intruder, and let V_A be the set of all variables that occur in the M_i (and are thus of type `Agent` in every SPS translation). Let $Subs$ be the set of substitutions from V_A to Ag. Thus $Subs$ represents all possible instantiations of the roles of the protocol with concrete agent names. If the SPS knowledge declarations contains some inequalities, such as $\mathtt{A} \neq \mathtt{p}$ or $\mathtt{A} \neq \mathtt{B}$, then this set $Subs$ is accordingly restricted.

Even though a knowledge itself is a substitution (cf. Definition 2), we now define what it means to apply a substitution (from $Subs$) to it. Let $\sigma \in Subs$ and $M = [\mathcal{X}_1 \mapsto t_1, \cdots, \mathcal{X}_l \mapsto t_l]$ be a knowledge. Then, we define $\sigma(M) = [\mathcal{X}_1 \mapsto \sigma(t_1), \cdots, \mathcal{X}_l \mapsto \sigma(t_l)]$. The initial state of the transition system is $(S_0; K_0; \emptyset)$ where:

$$S_0 = \bigcup_{i=1}^{k} \{\sigma(M_i)(steps_i.\mathsf{finished}(n)) \mid \sigma \in Subs, \sigma(R_i) \neq \mathtt{p}, n \in \mathbb{N}\}$$
$$K_0 = \bigcup_{i=1}^{k} \{\sigma(ul(M_i)) \mid \sigma \in Subs, \sigma(R_i) = \mathtt{p}\} \cup Ag$$

Here we use a new event $\mathsf{finished}(n)$ (for each $n \in \mathbb{N}$) to create a countable number of unique operational strands for each instance $\sigma \in Subs$. Note that we apply the instantiation σ first to the knowledge of the role, and the so instantiated knowledge to the entire operational strand. For instance, for the trivial operational strand $[\mathcal{X}_1 \mapsto \mathtt{A}, \mathcal{X}_2 \mapsto B, \mathcal{X}_3 \mapsto \mathtt{shk}(\mathtt{A}, \mathtt{B})] : \mathsf{fresh}\ \mathcal{X}_4.\mathsf{send}(\mathsf{insec}, \mathtt{scrypt}(\mathcal{X}_3, \mathcal{X}_4)$ and the instance $\sigma = [\mathtt{A} \mapsto \mathtt{a}, \mathtt{B} \mapsto \mathtt{p}]$, we get the countably many operational strands $[\mathcal{X}_1 \mapsto \mathtt{a}, \mathcal{X}_2 \mapsto \mathtt{p}, \mathcal{X}_3 \mapsto \mathtt{shk}(\mathtt{a}, \mathtt{p})] : \mathsf{fresh}\ \mathcal{X}_4.\mathsf{send}(\mathsf{insec}, \mathtt{scrypt}(\mathtt{shk}(\mathtt{a}, \mathtt{p}), X_4)).\mathsf{finished}(n).0$ for each $n \in \mathbb{N}$. All remaining variables in the instantiated operational strands represent freshly created values and (parts of) received messages.

Note that here we have created only the instances for the honest agents (because of the side condition $\sigma(R_i) \neq \mathtt{p}$); this is so since the behavior of the honest agents is subsumed by the abilities of the intruder when given the appropriate knowledge of the role in all instances where he plays the role.[8] With K_0 we

[8] In fact, we define here the semantics of operational strands using a standard Dolev-Yao style intruder deduction relation; stronger models could be employed, we just require that the intruder can at least perform the actions that honest agents can, i.e., encryption and decryption with known keys and the like.

therefore define the initial knowledge that the intruder needs to play in all roles under his real name. Here we model the *intruder knowledge* simply as a *set* of messages (rather than a substitution M as for honest agents) as for the standard Dolev-Yao intruder deduction, we do not need labels (and we do not consider notions like behavorial equivalence here). Accordingly, let $\vdash'$ denote the standard unlabeled intruder deduction on unlabeled messages, and the *ul* function mapping from a knowledge to a set of terms by discarding the labels. Thus, $ul([\mathcal{X}_1 \mapsto t_1, \cdots, \mathcal{X}_n \mapsto t_n]) = \{t_1, \cdots, t_n\}$ and $ul(M) \vdash' t$ iff $M \vdash t^l$ for some l.

The transition relation The transition relation $\Longrightarrow$ is defined as the least relation closed under the following rules:

$$\begin{array}{ll}
\text{T1} & (\{\mathsf{send}(\mathsf{insec}, t).\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\mathit{rest}\} \cup S; K \cup \{t\}; E) \\
\text{T2} & (\{\mathsf{receive}(\mathsf{insec}, t).\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup S; K; E) \\
 & \qquad \text{for any substitution } \sigma \text{ such that } K \vdash' \sigma(t) \\
\text{T3} & (\{\mathsf{event}(t).\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\mathit{rest}\} \cup S; K; E \cup \{\mathsf{event}(t)\}) \\
\text{T4} & (\{s \doteq t.\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\mathit{rest}\} \cup S; K; E) \qquad \text{if } s \approx t \\
\text{T5} & (\{\mathsf{fresh}\ X_i.\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup S; K; E) \\
 & \qquad \text{where } \sigma = [\mathcal{X}_i \mapsto c] \text{ and } c \text{ is a fresh constant} \\
\text{T6} & (\{\mathcal{X}_i := t.\mathit{rest}\} \cup S; K; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup S; K; E) \\
 & \qquad \text{where } \sigma = [\mathcal{X}_i \mapsto t] \\
\text{T7} & (\{0\} \cup S; K; E) \Longrightarrow (S; K; E)
\end{array}$$

The rules T1 and T2 handle the sending and receiving over an insecure channel: we add every sent message t to the intruder knowledge; for an agent who wants to receive a message of the *form* t (note that t may contain variables that are bound in this step), the intruder can use any instance $\sigma(t)$ that he can derive from his knowledge and we apply σ to the rest of the strand, i.e., instantiating all variables that have been bound in this step. We have only discussed the standard case of insecure channels here, other kinds of channels can be defined as in the ideal channel model of [**?**].

Note that the following invariants holds over all transitions: the intruder knowledge is a set of ground terms, all strands are closed, and all terms that the intruder can derive and send are thus also ground.

The other rules should be self-explanatory.

B Message Composition and Decomposition

In this section, we define the procedures for message composition and decomposition (*compose* and *analyze* respectively), but we first need some necessary definitions. First, we need to distinguish in the public operations between constructors and destructors. Note that the definitions and proofs of this section represent the details for the constructive proof of Theorem 2.

Definition 6. *Let* $\Sigma_d = \{\texttt{dscrypt}, \texttt{vscrypt}, \texttt{dcrypt}, \texttt{vcrypt}, \texttt{open}, \texttt{vsign}, \texttt{get}_., \texttt{verify}_.\}$ *be the* destructors*, where, abusing notation, we include* $\texttt{get}_.$ *and* $\texttt{verify}_.$ *for all formats. All other public operators* $\Sigma_c = \Sigma_p \setminus \Sigma_d$ *are called* constructors.

Let us also denote by $\approx_F$ the least congruence relation that satisfies properties (9)-(11) in Table 1 that address modular exponentiation and multiplication. Since we have here no destructors for `exp` and `mult`, $\approx_F$ is a *finite* theory; i.e., for any term t, the equivalence class of t under $\approx_F$ is finite (and moreover, unification is *finitary*, i.e., we can find finitely many most general unifiers for every pair of terms). We also define $\vdash_C$ as a restriction of $\vdash$ (Definition 3) where $\approx$ is replaced with $\approx_F$ and restricting Σ_p to Σ_c. Thus, $\vdash_C$ is the "compositional" part of the $\vdash$ relation that allows only composing terms and application of $\approx_F$ (which never "decomposes" terms).

B.1 Message Composition

We now define the compositional part of message deduction, i.e., computing $\vdash_C$, realized by the function $\mathit{compose}_M(t)$ that computes all labels for generating the term t from knowledge M using only $\vdash_C$.

Definition 7. *Let* M *be a knowledge and* $t \in \mathcal{T}_{\Sigma_A}(V)$.

$$\begin{aligned}\mathit{compose}_M(t) = \{\mathcal{X}_i \mid \exists t'.\ [\mathcal{X}_i \mapsto t'] \in M \wedge t \approx_F t'\} \cup \\ \{f(l_1,\ldots,l_n) \mid \exists t_1,\ldots,t_n.\ t \approx_F f(t_1,\ldots,t_n) \wedge f \in \Sigma_c \wedge \\ l_1 \in \mathit{compose}_M(t_1) \wedge \ldots \wedge l_n \in \mathit{compose}_M(t_n)\}\,.\end{aligned}$$

The first part of $\mathit{compose}_M$ checks whether the term t is directly contained in the knowledge modulo $\approx_F$, and returns corresponding label variables if so. The second part computes all ways to recursively compose t from its direct subterms (modulo $\approx_F$). For instance, for $M = [\mathcal{X}_1 \mapsto c, \mathcal{X}_2 \mapsto \texttt{hash}(c)]$ we have $\mathit{compose}_M(\texttt{hash}(c)) = \{\mathcal{X}_2, \texttt{hash}(\mathcal{X}_1)\}$, and for $M = [\mathcal{X}_1 \mapsto a \cdot b, \mathcal{X}_2 \mapsto c, \mathcal{X}_3 \mapsto a \cdot c, \mathcal{X}_4 \mapsto b]$ (writing $a \cdot b$ for $\texttt{mult}(a,b)$), we have $\mathit{compose}_M(a \cdot b \cdot c) = \{\mathcal{X}_1 \cdot \mathcal{X}_2, \mathcal{X}_3 \cdot \mathcal{X}_4\}$.

The $\mathit{compose}_M$ function does not involve any decomposition steps or generate checks—for this we define an analysis procedure in the next subsection. The interface between the two procedures is the notion of an *analyzed* knowledge (in which every possible analysis step has already been done). We define this notion succinctly by requiring that every term that can be derived from M using $\vdash$ can also be derived using $\vdash_C$, i.e., analysis steps do not yield any further messages:

Definition 8. *We say a knowledge M is* analyzed *iff*

$$\{t \in \mathcal{T}_{\Sigma_A}(V) \mid \exists\, l.\ M \vdash t^l\} = \{t \in \mathcal{T}_{\Sigma_A}(V) \mid \exists\, l.\ M \vdash_C t^l\}.$$

For an analyzed knowledge M, $compose_M$ is in fact correct:

Theorem 3 *The $compose_M$ function terminates and is sound in the sense that $l \in compose_M(t)$ implies $M \vdash t^l$. Moreover, if M is analyzed and neither M nor t contain symbols from Σ_d, then $compose_M$ is also complete in the sense that $M \vdash t^l$ implies $l' \in compose_M(t)$ for some label l' with $ccs(M) \models l \doteq l'$.*

Proof. For *termination*, consider the tree of recursive calls that $compose_M(t)$ can invoke. The tree is finitely branching since $\approx_F$ is a finite theory (every term has a finite equivalence class). Suppose the tree has infinite depth, and let $t_1, t_2, t_3, \ldots$ be the terms in the recursive calls. Then there are terms $t'_1, t'_2, t'_3, \ldots$ such that $t_i \approx_F t'_i \sqsupset t_{i+1}$ for all $i \geq 1$. Then there are contexts $C_1[\cdot], C_2[\cdot], \ldots$ and $C_i[x] \neq x$ such that $t_1 \approx_F C_1[t_2] \approx_F C_1[C_2[t_3]] \approx_F \ldots$ and thus t_1 has an infinite equivalence class modulo $\approx_F$, which is absurd, so the tree also has finite depth.

Soundness is immediate.

For *completeness*, consider $M \vdash t^l$, where M is analyzed and M and t do not contain any symbols from Σ_d. Since M is analyzed, we also have $M \vdash_C t^{l'}$ for some l', and thus $ccs(M) \models l \doteq l'$. Due to $\vdash_C$, l' cannot contain any symbol from Σ_d either (while l can). Consider now the proof tree for $M \vdash_C t^{l'}$: leaf nodes are axioms and inner nodes are either composition steps with $f \in \Sigma_c$ or algebraic equivalences modulo $\approx_F$. It is straightforward to map them into corresponding steps of $compose_M(t)$ to yield label l'.

B.2 Message Decomposition and Checks

To compute an analyzed knowledge and the checks that one can perform on it, we define the procedure *analyze* that takes as input a pair (M, φ) of a knowledge and a (finite) conjunction of equations and yields a *saturated* extension $(M \cup M', \varphi \wedge \varphi')$ of (M, φ). The notion of *saturated* means that $M \cup M'$ is analyzed and that $\varphi \wedge \varphi'$ is equivalent to $ccs(M \cup M')$. Note that this algorithm works incrementally, so when augmenting M with a received message in the generation of operational strands, we do not need to start the analysis from scratch. Also, we assume that we never add redundant checks, i.e., ones that are already entailed by previous checks.

Table 2 summarizes the procedure $analyze(M, \varphi)$. The table is divided into two parts: the upper part represents the first phase of the algorithm, saturating M with derivable subterms, whereas the lower part represents the second phase saturating φ with additional equations.

Phase 1. Here we check for every entry in M whether it can be analyzed, i.e., if it has one of the forms of column 1 (the head symbol being `scrypt`, `crypt`, `sign`, or a format) and that has not yet been marked as analyzed (initially no

1. Term of the form	2. Condition	3. Derive	4. Check	5. Recipe
$\mathcal{X}_i \mapsto \mathtt{scrypt}(k, m)$	$l \in \mathit{compose}_M(k)$	$\mathcal{X}_{i'} \mapsto m$	$\mathtt{vscrypt}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{dscrypt}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{crypt}(k, m)$	$l \in \mathit{compose}_M(\mathtt{inv}(k))$	$\mathcal{X}_{i'} \mapsto m$	$\mathtt{vcrypt}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{dcrypt}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{sign}(\mathtt{inv}(k), m)$	$l \in \mathit{compose}_M(k)$	$\mathcal{X}_{i'} \mapsto m$	$\mathtt{vsign}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{open}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{f}(t_1, \ldots, t_n)$ for some $\mathtt{f} \in \Sigma_f$	(true)	$\mathcal{X}_{i'_1} \mapsto t_1 \ldots$ $\mathcal{X}_{i'_n} \mapsto t_n$	$\mathsf{verify}_{\mathtt{f}}(\mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'_1} := \mathsf{get}_{1,\mathtt{f}}(\mathcal{X}_i) \ldots$ $\mathcal{X}_{i'_n} := \mathsf{get}_{n,\mathtt{f}}(\mathcal{X}_i)$

1. Term of the form ($i \neq j$)	2. Condition (where $\frac{t_d}{s_d} = \mathit{shorten}(\frac{t_1 \cdot \ldots \cdot t_n}{s_1 \cdot \ldots \cdot s_m})$)	3. Check
$\mathcal{X}_i \mapsto t$	$\{l_1, \ldots, l_n\} = \mathit{compose}_M(t)$	$l_1 \doteq l_2 \doteq \ldots \doteq l_n$
$\mathcal{X}_i \mapsto \mathtt{inv}(k)$	$l \in \mathit{compose}_M(k)$	$\mathtt{vcrypt}(\mathcal{X}_i, \mathtt{crypt}(l, \top)) \doteq \top$
$\mathcal{X}_i \mapsto t_1 \cdot \ldots \cdot t_n$ $\mathcal{X}_j \mapsto s_1 \cdot \ldots \cdot s_m$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$l_{t_d} \cdot \mathcal{X}_j \doteq l_{s_d} \cdot \mathcal{X}_i$
$\mathcal{X}_i \mapsto \mathtt{exp}(A, t_1 \cdot \ldots \cdot t_n)$ $\mathcal{X}_j \mapsto \mathtt{exp}(A, s_1 \cdot \ldots \cdot s_m)$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$\mathtt{exp}(\mathcal{X}_j, l_{t_d}) \doteq \mathtt{exp}(\mathcal{X}_i, l_{s_d})$
$\mathcal{X}_i \mapsto \mathtt{exp}(A, t_1 \cdot \ldots \cdot t_n)$ $\mathcal{X}_j \mapsto s_1 \cdot \ldots \cdot s_m$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_A \in \mathit{compose}_M(A)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$\mathtt{exp}(l_A, \mathcal{X}_j \cdot l_{t_d}) \doteq \mathtt{exp}(\mathcal{X}_i, l_{s_d})$

Table 2. Tabular overview of $\mathit{analyze}(M, \varphi)$

term is). We then check according to column 2 whether the necessary decryption key can be derived. For this, we use the $\mathit{compose}_M$ procedure yielding a label l if the key is available; if there are several labels, we simply pick one. We then mark the entry $\mathcal{X}_i \mapsto \ldots$ as analyzed, choose a new label variable $\mathcal{X}'_i$ and add the analyzed message to the knowledge M according to column 3. Further, we add the condition of column 4 and the recipe of column 5 to ϕ. (We treat the recipe here like an equation for simplicity.) We repeat this until no more analysis step can be performed. (Note: whenever new terms are added to M, encrypted messages that have not been marked as analyzed need to be checked again.)

Phase 2. We now consider every entry of M once and check for all alternative ways to generate it according to the first row in the lower part of the table. If we find more than one such label, we add the respective checks to ϕ. The second row is to check if a private key fits to its corresponding public key if it is known.[9]

Next, we have rules for products and exponents (last three rows of the lower table). Here we consider any pair of entries in M where the head symbol is `exp` or `mult` (according to the form of column 1), again writing $\cdot$ for multiplication. Here, we require a match such that none of the s_i and t_i is itself a product. We then consider the fraction $(t_1 \cdot \ldots \cdot t_n)/(s_1 \cdot \ldots \cdot s_m)$ and *shorten* it, i.e., removing common factors in enumerator and denominator. Let t_d/s_d be remaining products after shortening. If all the t_i or all the s_i are shortened away (i.e., $t_d = 1$

[9] This check is actually quite academic, as the agent has either generated the key pair itself (and thus knows by construction that they form a key pair) or it has received it from a key server, e.g., in identity-based encryption (but then needs to trust that server anyway). However, without this check the correctness theorem and its proof would require a more complicated formulation.

or $ts_d = 1$) we do not apply this rule (as it is already covered by the first row, saving us from introducing 1 into the algebraic theory). We now try to compose the products t_d and s_d according to column 2. If there is at least one label for each (if there are several, again we pick one), then we add to ϕ the condition of column 3.

Example 3. We compute $\mathit{analyze}(M, \top)$ for the knowledge $M = [\mathcal{X}_1 \mapsto y, \mathcal{X}_2 \mapsto \mathtt{scrypt}(\mathtt{exp}(\mathtt{exp}(g, y), x), n), \mathcal{X}_3 \mapsto \mathtt{scrypt}(k, \mathtt{exp}(g, x)), \mathcal{X}_4 \mapsto k, \mathcal{X}_5 \mapsto \mathtt{hash}(n)]$.

For phase 1, entries $\mathcal{X}_1$, $\mathcal{X}_4$, and $\mathcal{X}_5$ do not match any entry in the first column (they cannot possibly be decrypted). For $\mathcal{X}_2$, we have $\mathit{compose}_M(\mathtt{exp}(\mathtt{exp}(g, y), x)) = \emptyset$, i.e., the decryption key is not (yet) available. However, we can decrypt $\mathcal{X}_3$ since $\mathit{compose}_M(k) = \{\mathcal{X}_4\}$. We thus add $\mathcal{X}_6 \mapsto \mathtt{exp}(g, x)$ to the knowledge, and to ϕ the check $\mathtt{vscrypt}(\mathcal{X}_4, \mathcal{X}_3) \doteq \top$ and the recipe $\mathcal{X}_6 := \mathtt{dscrypt}(\mathcal{X}_4, X_3)$. We mark $\mathcal{X}_3$ as analyzed, and check again the unanalyzed $\mathcal{X}_2$. This time (for the updated M) we have $\mathit{compose}_M(\mathtt{exp}(\mathtt{exp}(g, y), x)) = \{\mathtt{exp}(\mathcal{X}_6, \mathcal{X}_1)\}$, and thus add $\mathcal{X}_7 \mapsto n$ to the knowledge, and to ϕ the check $\mathtt{vscrypt}(\mathtt{exp}(\mathcal{X}_6, \mathcal{X}_1), \mathcal{X}_2) \doteq \top$ and recipe $\mathcal{X}_7 := \mathtt{dscrypt}(\mathtt{exp}(\mathcal{X}_6, \mathcal{X}_1), \mathcal{X}_2)$. Since neither $\mathcal{X}_6$ and $\mathcal{X}_7$ can be further analyzed, phase 1 is finished. For phase 2, we can of course re-construct the encryptions, e.g., $\mathtt{scrypt}(\mathcal{X}_4, \mathcal{X}_6) \doteq \mathcal{X}_3$ but that is already implied by the equation $\mathcal{X}_6 := \mathtt{dscrypt}(\mathcal{X}_4, \mathcal{X}_3)$ and we do not add redundant checks. The only new check is for $\mathcal{X}_5$, since $\mathit{compose}_M(\mathtt{hash}(n)) = \{\mathcal{X}_5, \mathtt{hash}(\mathcal{X}_7)\}$ yields $\mathcal{X}_5 \doteq \mathtt{hash}(\mathcal{X}_7)$.

As another example for equational reasoning, $\mathit{analyze}([\mathcal{X}_1 \mapsto a \cdot b \cdot c, \mathcal{X}_2 \mapsto a \cdot c \cdot d, \mathcal{X}_3 \mapsto b, \mathcal{X}_4 \mapsto d], \top)$ yields the check $\mathcal{X}_1 \cdot \mathcal{X}_4 \doteq \mathcal{X}_2 \cdot \mathcal{X}_3$. □

Theorem 4 *For a knowledge M with no symbols in Σ_d and a finite conjunction ϕ of equations, $\mathit{analyze}(M, \phi)$ terminates with a result (M', ϕ') where $M' = M \cup [\mathcal{X}_{|M|+1} \mapsto t_1, \cdots, \mathcal{X}_{|M|+n} \mapsto t_n]$, $\phi' = \phi \wedge \mathcal{X}_{|M|+1} := l_1 \wedge \cdots \wedge \mathcal{X}_{|M|+n} := l_n \wedge \psi$ and $M \vdash t_1^{l_1}, \cdots, M \vdash t_n^{l_n}$ such that $\{t \mid M \vdash t^l\} = \{t \mid M' \vdash t^l\}$ (soundness), $\mathit{analyzed}(M')$ (completeness), and $\mathit{ccs}(M) \equiv \phi \wedge \psi$ (correctness of checks).*

Proof. Soundness is immediate, as we merely add derivable messages to the knowledge. *Termination:* The newly added terms of M' are always subterms of some term in M, so the M' component must eventually reach a fixed point. Adding new equations to ϕ' is bounded by pairs of entries of M' and the finiteness of $\mathit{compose}_M$.

Completeness: We first show that M' is analyzed, i.e., we have to show that for any term $t \in \mathcal{T}_{\Sigma_A}(V)$ with $M' \vdash t^l$, we also have $M' \vdash_C t^{l'}$ for some l' (i.e., using only constructors of Σ_c and equivalence in $\approx_F$). For this, we consider the proof tree for $M' \vdash t$. Intermediate nodes may well contain destructors, but we can exclude so-called *garbage terms*, namely terms that are not $\approx$-equivalent to any term in $\mathcal{T}_{\Sigma_A}(V)$. For instance, $\mathtt{dscrypt}(c, c)$ is garbage (while $\mathtt{dscrypt}(k, \mathtt{scrypt}(k, m)) \approx m$ is not). Suppose the proof contains a node with a garbage term s, then there must be a construction in the proof to remove s (since the final term must be in $\mathcal{T}_{\Sigma_A}(V)$), for instance constructing $\mathtt{dscrypt}(s, \mathtt{scrypt}(s, m)) \approx m$ eliminates garbage s, but in all such cases,

all occurrences of s must have been composed, so there exists a simpler proof without garbage.

We thus first show the following: for any $M' \vdash t^l$ where t is not garbage we have $M' \vdash_C s^k$ for some $s \approx t$ and some label k. This is shown by induction over the proof tree of $M' \vdash t^l$. For Ax and Eq the proof is immediate as well as for Cmp with $f \in \Sigma_c$. For $f \in \Sigma_d$, consider the term t_0 being analyzed. By induction $M' \vdash_C s_0^{k_0}$ for some $s_0 \approx t_0$, so this is (modulo $\approx_F$) either composed or an axiom. If it is composed, then the intruder decomposes a term he has composed himself and this proof can be simplified. If it is an axiom, then the intruder applies decomposition to a term in his knowledge, and *analyze* has already added the resulting term t (modulo $\approx$) to M'.

Note we have only proved that for $M' \vdash t^l$ (where t is not garbage) there is $M' \vdash_C s^k$ for some $s \approx t$. We have to show that $M' \vdash_C t^{l'}$ for some l', but only for $t \in \mathcal{T}_{\Sigma_A}(V)$, i.e., without destructors. We claim that in this case we have $s \approx_F t$ (and thus follows $M' \vdash_C t^k$ as $\vdash_C$ is closed under $\approx_F$). This claim follows from the fact that our destructor equations (1)–(8) can be read as rewrite rules (from left to right) that are convergent modulo $\approx_F$, and thus terms that do not contain constructors are in normal form modulo $\approx_F$. The idea for proving this convergence is that the rewriting rules have disjoint symbols from the equations in $\approx_F$ (so they cannot conflict) and we can prove convergence for the rewrite rules using the critical pair method, see e.g. [**?**].

Correctness of Checks: Now for $ccs(M) \equiv \phi \wedge \psi$. For brevity let $\psi' = \phi \wedge \psi$. The soundness ($ccs(M) \implies \psi'$) is obvious by checking that each step in *analyze* adds only sound equations. The completeness we prove again indirectly, i.e., suppose we have a term t and two derivation proofs $M \vdash t^l$ and $M \vdash t^{l'}$ such that $l \doteq l'$ is not implied by ψ'. Suppose in either of the derivation trees for l and l' appears a composition step with a destructor. Suppose the message being decomposed is $t_1^{l_1}$ and the result of decomposing is $t_0^{l_0}$. Again assume that there are no decompositions in the subtrees. One possible case is the analysis of `inv` which is covered by the sixth case in *analyze* (Table 2). In all other cases, $analyze(M, \phi)$ must have added t_0 under some new label X_i to M' and ψ' must entail $X_i \doteq l_0$ (and a constraint about verifiability of l_1). Let us thus replace the derivation $t_0^{l_0}$ with $t_0^{X_i}$: this changes a subterm in labels l and l', but for these changed labels still $l \doteq l'$ does not follow from ψ'. In this way we can step by step eliminate all analysis steps and thus have two trees without analysis for t^l and $t^{l'}$ such that $l \doteq l'$ is not implied by ψ'.

Now we consider the case that either of the two trees (say for l') is an application of only axiom and equality steps, thus $l' = X_i$ for some variable X_i. Then M' contains $[X_i \mapsto t]$ for a term that can be composed in a different way using only constructors and $\approx_F$, i.e., $l \in compose_M(t)$ and thus ψ' must contain $l \doteq l'$, contradicting the assumption. Otherwise it must be two trees consisting of composition steps. We can exclude composition with any operator but `exp` or `mult` since otherwise we can simply go to one of the subterms. If it is `exp` or `mult`, then it has the form of adding factors to initially known `exp` or `mult` terms. Again we can exclude adding the same factor in both trees (since otherwise we

can reduce again to a simpler case). The remaining case is however covered by our check rules for `exp` and `mult`, again showing that $l \approx l'$ must by entailed by ψ'.

Theorem 5 *For our example theory in Table 1, for every strand S in which no destructors or verifiers occur, $[\![S]\!]_L$ is recursive and has a finite representation.*

Proof. First, by Theorem 3, *compose* is recursive and produces a finite set of labels. Second, by Theorem 4, *analyze* is recursive and produces a finite conjunction of checks and a finite knowledge. Finally, given a strand S that is finite (being defined by context-free grammar), one can easily see from the rules of $[\![\cdot]\!]_L$ and from the previous two points that $[\![\cdot]\!]_L$ is recursive and it has a finite representation.

B.3 Equivalence of Strands

So far we defined the high-level semantics for SPS ($[\![\cdot]\!]_H$) that is succinct, simple and general, but as Theorem 1 states it is not recursive in general. To address that we defined a low-level semantics ($[\![\cdot]\!]_L$) that we proved in Theorem 5 that it is computable for a fixed set of operators that are representative for a wide range of real world protocols.**FIX**[10] The missing point now is the connection between the two semantics, i.e., given the same plain strand, whether they produce equivalent operational strands. We thus need to define a notion of *equivalence* between strands. Intuitively, two strands are equivalent if they have the same initial knowledge, corresponding send and receive steps, equivalent checks and events. Based on this notion of equivalence, we now discuss the rules of the two semantics to show that they produce equivalent operational strands.

– The initial case:

$$[\![M : strand]\!]_{H_0} = M : ccs(M).[\![strand]\!]_H(M)$$

with the corresponding rule:

$$[\![M : strand]\!]_{L_0}(\emptyset, \top) = M : \varphi.[\![strand]\!]_L(M', \varphi),$$

where $(M', \varphi) = analyze(M, \top)$.
The first difference between the results of the two rules is the checks, i.e., in the first one we have $ccs(M)$ and in the second one we have φ that is the conjunction of checks that *analyze* procedure produce. By Theorem 4 we have that they are equivalent, i.e., $ccs(M) \equiv \varphi$ in our case. The second difference is the knowledge carried on for the next steps, i.e., in the first rule we have M while in the second rule we have M' that is an analyzed version of M. Recall that a knowledge and its analyzed version are equivalent in a sense that one can derive the same terms from both. The main difference

[10] **OM FIX:** This introduction is a kind of a summary to help the reader to stay connected, is this fine?

between the two versions of a knowledge (the original and the analyzed) is that the analyzed version has no further analysis steps and this is needed for the termination of *compose* (cf. Definitions 7 and 8). **FIX**[11]

– send case:

$$[\![\mathsf{send}(ch,t).rest]\!]_H(M) = \mathsf{send}(ch,l).[\![rest]\!]_H(M),$$

where $M \vdash t^l$ for some label l, with the corresponding rule:

$$[\![\mathsf{send}(ch,t).rest]\!]_L(M,\varphi) = \mathsf{send}(ch,l).[\![rest]\!]_L(M,\varphi) \text{ where } l \in compose_M(t)$$

The only difference between the two rules is the way the recipe l is derived, i.e., in the first rule we have $M \vdash t^l$ and in the second rule we have $l \in compose_M(t)$. By Theorem 3 (soundness and completeness of *compose*) and by Theorem 4 we have that we either have the same label for t, or if we have different labels then a check must be added to reflect that (cf. Definition 4).

– event case: similar to the send case.

– receive case:

$$[\![\mathsf{receive}(ch,t).rest]\!]_H(M) = \mathsf{receive}(ch,\mathcal{X}_{|M|+1}).ccs(M \cup [\mathcal{X}_{|M|+1} \mapsto t]).\\ [\![rest]\!]_H(M \cup [\mathcal{X}_{|M|+1} \mapsto t])$$

with the corresponding rule:

$$[\![\mathsf{receive}(ch,t).rest]\!]_L(M,\varphi) = \mathsf{receive}(ch,\mathcal{X}_{|M|+1}).\varphi'.[\![rest]\!]_L(M',(\varphi \wedge \varphi'))\,,$$

where $(M', \varphi \wedge \varphi') = analyze(M \cup [\mathcal{X}_{|M|+1} \mapsto t], \varphi)$.
The difference between the two rules is the check parts again, but as we have in the initial case that by Theorem 4, $ccs(M) \equiv \varphi$. Note that the labels could be different (because an analyzed knowledge has in general more terms than the original version of it, cf. *analyze* procedure), so the result of these two rules may not be identical as they may be receiving the term t with different labels, but a proper α-renaming can resolve that and make the two resulting strands identical except that the knowledge of one of them is the analyzed version of the other that are in principle equivalent (one can derive the same terms from both).

– fresh case:

$$[\![\mathsf{fresh}\ X.rest]\!]_H(M) = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![rest]\!]_H(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\})$$

with the corresponding rule:

$$[\![\mathsf{fresh}\ X.rest]\!]_L(M,\varphi) = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![rest]\!]_L(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\},\varphi)$$

Again, the only difference that may occur between the two is the label of the fresh value X, but as we discussed in the previous case, a proper α-renaming can resolve it with no semantical effect.

By this, we can conclude that for our example theory in Table 1, $[\![\cdot]\!]_L$ is an implementation of $[\![\cdot]\!]_H$.

[11] **Luca FIX:** Weird to cite them in different order **OM** Done

C Translating to Applied-π

Here, we present $[\![\cdot]\!]_\pi$ that translates an operational strand to an applied-π calculus process. We use the syntax provided in [**?**]. Note that the semantics of operational strands is actually similar to a process calculus and this translation to it is mainly a matter of pretty-printing, yet some details that we explain shortly. We define $[\![]\!]_\pi$ as follows (+ denotes string concatenation):

$[\![M{:}strand]\!]_\pi$ = `let proc_`+$own(strand)$+ "("+ $par(M)$+")=" + $[\![$strand$]\!]_\pi$
where: $own(strand)$ is the name of the agent that owns the strand $strand$,
and $par(M)$ is a list of the process parameters derived from its initial knowledge M

$[\![\mathsf{send}(ch, l).rest]\!]_\pi$ = "`out`("+ ch + ","+ l+ ");"+ $[\![rest]\!]_\pi$

$[\![\mathsf{receive}(ch, l).rest]\!]_\pi$ = "`in`("+ ch + ","+ l+ ");"+ $[\![rest]\!]_\pi$

$[\![\mathsf{fresh}\ l.rest]\!]_\pi$ = "`new` " + l+ ":`bitstring`;"+ $[\![$rest$]\!]_\pi$

$[\![x := t.rest]\!]_\pi$ = "`let` "+ x+ "="+ t+ "`in` "+ $[\![rest]\!]_\pi$

$[\![t \doteq \top.rest]\!]_\pi$ = $[\![rest]\!]_\pi$

$[\![s \doteq t.rest]\!]_\pi$ = "`if`("+ s+ "="+ t ") `then` " + $[\![rest]\!]_\pi$

$[\![\mathsf{event}(t).rest]\!]_\pi$ = "`event`(t);" + $[\![rest]\!]_\pi$

$[\![0]\!]_\pi$ = "0."

The first rule declares the agent's process; by giving it a name and parametrize it over the initial knowledge of the agent. For example, Let $M^A : strand^A$ be the strand shown in our example in Figure 1(b), then $own(strand^A) = \mathtt{A}$, and $par(M^A) = \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{x4} : \mathtt{bitstring}$, so the process will be `proc_A` as shown in the first line of the second column of Figure 2. The second and the third rules deal with the sending and receiving of messages over a channel ch. The forth rule deals with the creation of a fresh value, and the fifth rule covers the macro case of a strand and how it is translated in applied π code. The sixth and the seventh rules deal with the checks. Note that in the case of a check that one of its sides is the true value $\top$, we simply ignore such case since this check is implicitly performed by the next destruction step. For example, consider the translation of Figure 1(b), we ignore $\mathtt{vscrypt}(\mathcal{X}_3, \mathcal{X}_6) \doteq \top$ as it is followed by $\mathcal{X}_7 := \mathtt{dscrypt}(\mathcal{X}_3, \mathcal{X}_6)$, which according to the property (`reduc forall` $\mathtt{m}, \mathtt{k} : \mathtt{bitstring}$; $\mathtt{dscrypt}(\mathtt{k}, \mathtt{scrypt}(\mathtt{k}, \mathtt{m})) = \mathtt{m}$.) will not be decrypted $\mathcal{X}_6$ unless it is a valid encrypted message and $\mathcal{X}_3$ is a valid encryption key. The eighth rule pretty-prints the event $\mathsf{event}(t)$ in the process and the last rule ends the strand.

D Translating to JavaScript

We define the function $[\![\cdot]\!]_{JS}$ that translates from operational strand to JavaScript code. In the definition below we use + for string concatenation.

$$
\begin{array}{ll}
[\![M : steps]\!]_{JS} & = head(M : steps) + [\![steps]\!]_J \\
[\![\mathsf{receive}(ch, \mathcal{X}_i).rest]\!]_{\not\mathcal{F}} & = \text{“}\texttt{byte[ ] } \text{”} + \mathcal{X}_i + \text{“} = ch.\mathsf{receive}();\text{”} + [\![rest]\!]_J \\
[\![\mathsf{send}(ch, l).rest]\!]_J & = \text{“}ch.\mathsf{send}(\text{”} + l + \text{“});\text{”} + [\![rest]\!]_J \\
[\![\mathsf{verify}_{\mathtt{f}}(l) \doteq \top.rest]\!]_{\not\mathcal{F}} & = \texttt{f} + \text{“ ”} + l + \text{“}a = \texttt{new } \text{”} + \texttt{f} + \text{“}(\text{”}. + l + \text{“});\text{”} \\
 & \quad + [\![rest]\!]_J \\
[\![\mathcal{X}_i := \mathsf{get}_{i,\mathtt{f}}(l).rest]\!]_{\not\mathcal{F}} & = \text{“}\texttt{byte[ ] } \text{”} + \mathcal{X}_i + \text{“} = \text{”} + l + \text{“}a.get\text{”} + i + \text{“}();\text{”} \\
 & \quad + [\![rest]\!]_J \\
[\![X_i := t.rest]\!]_J & = \text{“}\texttt{byte[ ] } \text{”} + \mathcal{X}_i + \text{“} = \text{”} + t + \text{“};\text{”} + [\![rest]\!]_J \\
[\![t \doteq \top.rest]\!]_J & = [\![rest]\!]_J \\
[\![t \doteq s.rest]\!]_J & = \text{“}\texttt{if}(\text{”} + t + \text{“}! = \text{”} + s + \text{“}) \texttt{ error}(); + [\![rest]\!]_J \\
[\![0]\!]_J & = \text{“}\}\text{”}
\end{array}
$$

where: $head(M : steps) =$ “`void proc_`”$+own(M : steps)+$“(”$+par(M)+$“){”, $own(M : steps)$ is the agent that owns the strand $M : steps$, and $par(M)$ is the knowledge M formed as a list of parameters, i.e., a comma separated list of the label variables of the knowledge M. We add to this list a channel object ch given as additional parameter that the code uses to send and receive messages as we explain later. In a nutshell, the first rule gives the header of the JavaScript code that we want to generate from the operational strand S. For example, let S be the operational strand given in Fig. 1(b), then $head(S) =$ `function proc_A(X1, X2, X3, X4, ch){`. The left bracket that we have at the end starts the function.

In the receive rule, we declare a new variable $\mathcal{X}_i$ of type `byte []` (byte string). We then assign to $\mathcal{X}_i$ the value received from the channel ch via the method receive(), i.e., the value obtained from ch.receive() is assigned to that variable $\mathcal{X}_i$. Next, the send rule uses the method send() to send the term l over the channel ch . Recall that l is a recipe that tells the code how to construct some value in reference to the given parameters, received messages, or derived messages.

The third rule handles a special case of checks, namely the case of a format verifier ($\mathsf{verify}_{\mathtt{f}}(l) \doteq \top$) where l represents what is supposed to be a “serialized” `f` object, i.e., a byte string that is supposed to be parsed as an object of type `f`. In this case, we create an object of that format where the name of the object is obtained simply by appending the letter ‘a’ to the string name l (there is no significance in choosing the letter ‘a’, it is just that we need to distinguish between the byte string called l and the new object that we need to create when we parse l, so we called the object $l +$ “a”). Note that The reason behind the creation of an object is of twofold. First, we need the created object for later use (in obtaining the different fields of the formatted message as we see in the next rule); for that we can not directly use the byte string l directly. Second, we need to verify that the string l is indeed of the format `f`. This verification is not explicitly done, instead it is left to the constructor that maps l to an object of type `f`. The constructor here is basically a format parser. The next rule is dedicated for macros in which format getters are involved. Recall that by a format getter we

mean the format methods that obtain different fields of a format object. This is achieved simply by calling the `get` method of the format object that we created when we encountered the format verifier ($\texttt{verify}(l)$). Here we rely on the fact that our model generates a format verifier before decomposing a format. Note that this case is a special case of macros, the next rule ($[\![X_i := s.rest]\!]_J$) handles the other cases of macros. When we have the macro $\mathcal{X}_i := t$, we simply create a new variable $\mathcal{X}_i$ and give it the value t.

In the seventh rule we handle another special case of checks, namely the case with operator verifiers except the format verifiers (we already handled format verifiers in the third rule). The operator verifiers that we have in SPS as we discussed earlier are $\{\texttt{vcrypt}, \texttt{vsign}, \texttt{vscrypt}\}$. In this case, we do not produce any JavaScript code; simply because the verification is left to the deconstructors**FIX**[12] $\{\texttt{dcrypt}, \texttt{open}, \texttt{dscrypt}\}$; since they implement a verification mechanism, e.g., the decryption will raise an exception if it failed to decrypt a supposedly encrypted message. Here, we rely on the fact that our model calls the decryption step immediately after the verification (cf. Table **??**), so we delay the verification to the decryption that implements it. The eighth rule is for the other cases of checks, namely when we require that two terms (none of them is $\top$) are equal. We translate this check into an if-statement, i.e., if the two labels of the check are not equal then we arise an error. The last rule handles the end of the operational strand that we translate to a right bracket (}) closing the left bracket we opened in the very fist rule (by calling $head(\cdot)$).

[12] **OM FIX:** I do not want to say destructors to avoid any ambiguity with class destructors, any suggestions?